1985

The design and implementation of a relational interface for the multi-lingual database system

Kloepping, Gary R.; Mack, John F.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DESIGN AND IMPLEMENTATION OF
A RELATIONAL INTERFACE FOR
THE MULTI-LINGUAL DATABASE SYSTEM

by
Gary R. Kloepping
and
John F. Mack

June 1985

Thesis Advisor: David K. Hsiao

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
The Design and Implementation of A Relational Interface For the Multi-Lingual Database System

5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis
June 1985

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Gary R. Kloepping and
John F. Mack

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, CA 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, CA 93943

12. REPORT DATE
June 1985

13. NUMBER OF PAGES
184

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Multi-Lingual Database System; Multi-Backend Data Base System; Relational Database; Relational/SQL Interface

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
Traditionally, the design and implementation of a conventional database system begins with the choice of a data model followed by the specification of a model-based data language. Thus, the database system is restricted to a single data model and a specific data language. An alternative to this traditional approach to database-system development is the multi-lingual database system (MLDS). This alternative approach enables the user
(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

to access and manage a large collection of databases via several data models and their corresponding data language without the aforementioned restriction.

In this thesis, we present the specification and implementation of a relational/SQL language calls into attribute-based date language (ABDL) requests. We describe the software engineering aspects of our implementation and an overview of the four modules which comprise our relational/SQL language interface.

Approved for Public Release, Distribution Unlimited.

The Design and Implementation of a Relational Interface for the Multi-Lingual Database System

by

Gary R. Kloepping
Captain, United States Army
B.S., United States Military Academy, 1976

and

John F. Mack
Captain, United States Army
B.S., United States Military Academy, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

ABSTRACT

Traditionally, the design and implementation of a conventional database system begins with the choice of a data model followed by the specification of a model-based data language. Thus, the database system is restricted to a single data model and a specific data language. An alternative to this traditional approach to database-system development is the multi-lingual database system (MLDS). This alternative approach enables the user to access and manage a large collection of databases via several data models and their corresponding data languages without the aforementioned restriction.

In this thesis we present the specification and implementation of a relational/SQL language interface for the MLDS. Specifically, we present the specification and implementation of an interface which translates SQL language calls into attribute-based data language (ABDL) requests. We describe the software engineering aspects of our implementation and an overview of the four modules which comprise our relational/SQL language interface.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. MOTIVATION

During the past twenty years database systems have been designed and implemented using what we refer to as the traditional approach. The first step in the traditional approach involves choosing a data model. Candidate data models include the relational data model, the hierarchical data model, the network data model, the entity-relationship data model, or the attribute-based data model to name a few. The second step specifies a model-based data language, e.g., SQL or QUEL for the relational data model, or Daplex for the entity-relationship data model.

A number of database systems have been developed using this methodology. For example, there is IBM's Information Management System (IMS) since the sixties, which supports the hierarchical data model and the hierarchical-model-based data language, Data Language I (DL/I). Sperry Univac has introduced the DMS-1100 in the early seventies, which supports the network data model and the network-model-based data language, CODASYL Data Manipulation Language (CODASYL-DML). And more recently, there has been IBM's introduction of the SQL/Data System

which supports the relational model and the relational-model-based data language, Structured English Query Language (SQL). The result of this traditional approach to database system development is a homogeneous database system that restricts the user to a single data model and a specific model-based data language.

An unconventional approach to database system development, referred to as the Multi-lingual database system (MLDS) [Ref. 1], alleviates the aforementioned restriction. This new system affords the user the ability to access and manage a large collection of databases via several data models and their corresponding data languages. The design goals of MLDS involve developing a system that is accessible via a relational/SQL interface, an hierarchical/DL/I interface, a network/CODASYL interface, and an entity-relationship/Daplex interface.

There is a number of advantages in developing such a system. Perhaps the most practical of these involves the reusability of database transactions developed on an existing database system. In MLDS, there is no need for the user to convert a transaction from one data language to another data language. The MLDS permits the running of database transactions written in different data languages. Hence, the user does not have to perform either a manual or automated translation of an existing transaction in order to execute the transaction in MLDS.

The MLDS provides the same results even if the data language of the transaction is originated at a different database system.

A second advantage deals with the economy and effectiveness of hardware upgrade. Frequently, the hardware supporting the database system is upgraded because of technological advancements or system demand. With the traditional approach, this type of hardware upgrade has to be provided for all of the different database systems in use, so that all of the users can experience system performance improvements. This is not the case in MLDS, where only the upgrade of a single system is necessary. In a MLDS, the benefits of a hardware upgrade are uniformly distributed across all users, despite their use of different models and data languages.

Thirdly, a multi-lingual database system allows users to explore the desirable features of the different data models and then use these to better support their applications. This is possible because MLDS supports a variety of databases structured in any of the well-known data models.

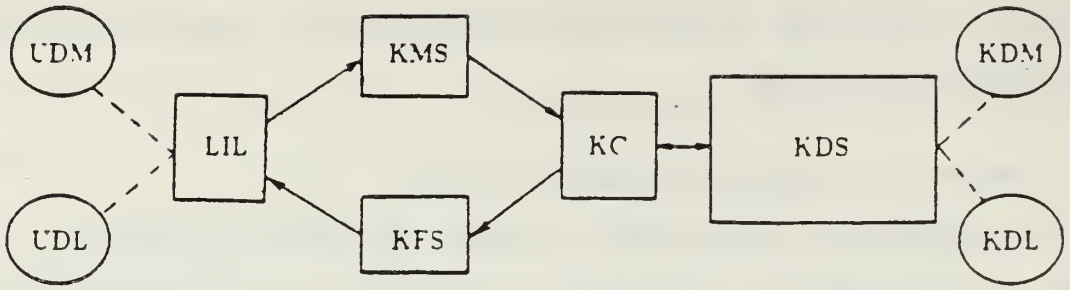
It is apparent that there exists ample motivation to develop a multi-lingual database system with many data model/data language interfaces. In this thesis, we are developing a relational/SQL language interface for the MLDS. We are extending the work of Macy [Ref. 2] and Rollins

[Ref. 3], who have showed the feasibility of this particular interface in a MLDS.

B. THE MULTI-LINGUAL DATABASE SYSTEM

A detailed discussion of each of the components of a MLDS is provided in subsequent chapters. In this section we provide an overview of the organization of a MLDS. This can assist the reader in understanding how the different components of the MLDS are related.

Figure 1 shows the system structure of a multi-lingual database system. The user interacts with the system through the language interface layer (LIL) using a chosen user data model (UDM) to issue transactions written in a corresponding model-based user data language (UDL). The LIL routes the user transactions to the kernel mapping system (KMS). The KMS performs one of two possible tasks. First, the KMS transforms a UDM-based database definition to a database definition of the kernel data model (KDM) when the user specifies that a new database is to be created. When the user specifies that a UDL transaction is to be executed, the KMS translates the UDL transaction to a transaction in the kernel data language (KDL). In the first task, the KMS forwards the KDM data definition to the kernel controller (KC). The KC, in turn, sends the KDM database definition to the kernel database system (KDS). When the KDS is finished with processing



UDM : User Data Model
UDL : User Data Language
LIL : Language Interface Layer
KMS : Kernel Mapping System
KC : Kernel Controller
KFS : Kernel Formatting System
KDM : Kernel Data Model
KDL : Kernel Data Language
KDS : Kernel Database System

Figure 1. The Multi-Lingual Database System.

the KDM database definition, it informs the KC. The KC then notifies the user via the LIL that the database definition has been processed and that the loading of the database records may begin. In the second task, the KMS sends the KDL transactions to the KC. When the KC receives the KDL transactions, it forwards them to the KDS for execution. Upon completion, the KDS sends the results in KDM form back to the KC. The KC routes the results to the kernel formatting system (KFS). The KFS reformats the results from KDM form to UDM form. The KFS then displays the results in the correct UDM form via the LIL.

The four modules, LIL, KMS, KC, and KFS, are collectively known as the language interface. Four similar

modules are required for each other language interface of the MLDS. For example, there are four sets of these modules where one set is for the relational/SQL language interface, one for the hierarchical/DL/I language interface, one for the network/CODASYL language interface, and the last one for the entity-relationship/Daplex language interface. However, if the user writes the transaction in the native mode, i.e. in KDL, there is no need of any interface.

In our implementation of the relational/SQL language interface, we develop the code for the four modules. However, we do not integrate these modules with the KDS as shown in Figure 1. The Laboratory of Database Systems Research at the Naval Postgraduate School is in the process of procuring new computer equipment for the KDS. When the equipment is installed, the KDS will be ported over to the new equipment. The MLDS software will then be integrated with the KDS. Although not a very difficult undertaking, it may be time consuming.

C. THE KERNEL DATA MODEL AND LANGUAGE

The choice of a kernel data model and a kernel data language is the key decision in the development of a multi-lingual database system. The overriding question, when making such a choice, is whether the kernel data model and kernel data language is capable of

supporting the required data-model transformations and data-language translations for the language interfaces.

The attribute-based data model proposed by Hsiao [Ref. 4], extended by Wong [Ref. 5], and studied by Rothnie [Ref. 6], along with the attribute-based data language (ABDL), defined by Banerjee [Ref. 7], have been shown to be acceptable candidates for the kernel data model and kernel data language, respectively.

Why is the determination of a kernel data model and kernel data language so important for a MLDS? No matter how multi-lingual the MLDS may be, if the underlying database system (i.e., KDS) is slow and inefficient, then the interfaces may be rendered useless and untimely. Hence, it is important that the kernel data model and kernel language be supported by a high-performance and great-capacity database system. Currently, only the attribute-based data model and the attribute-based data language are supported by such a system. This system is the multi-backend database system (MBDS) [Ref. 1].

D. THE MULTI-BACKEND DATABASE SYSTEM

The multi-backend database system (MBDS) has been designed to overcome the performance problems and upgrade issues related to the traditional approach of database system design. This goal is realized through the utilization of multiple backends connected in a parallel

fashion. These backends have identical hardware and replicated software and their own disk systems. In a multiple backend configuration, there is a backend controller, which is responsible for supervising the execution of database transactions and for interfacing with the hosts and users. The backends perform the database operations with the database stored on the disk system of the backends. The controller and backends are connected by a communication bus. Users access the system through either the hosts or the controller directly (See Figure 2).

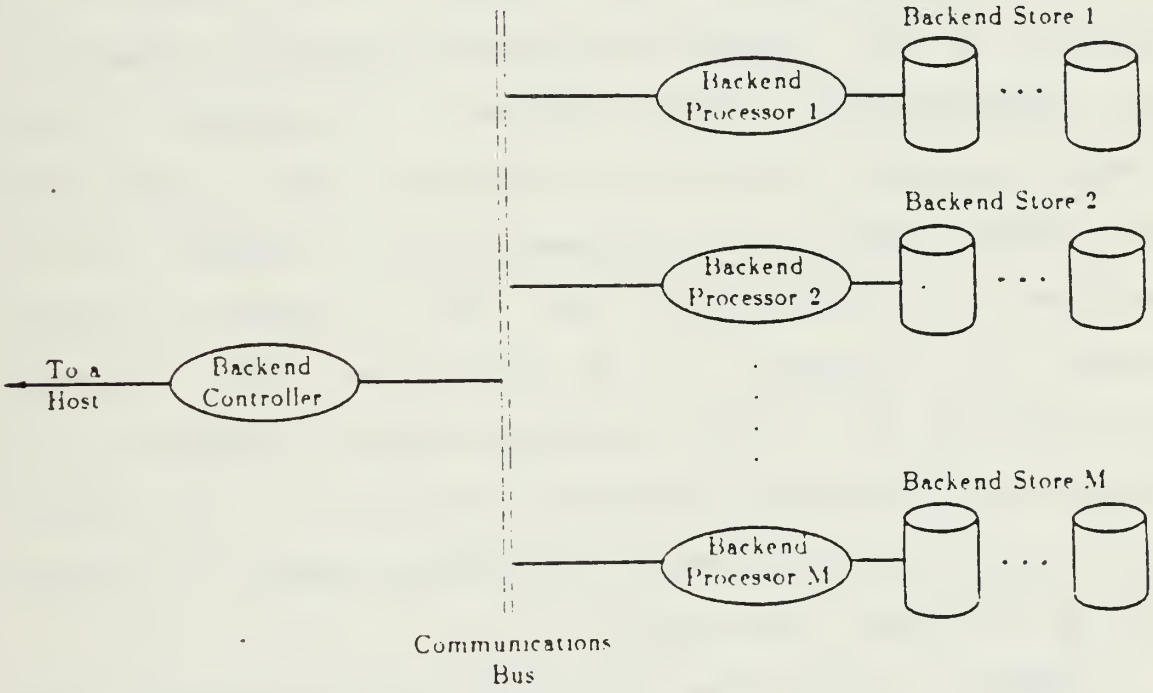


Figure 2. The Multi-Backend Database System.

Performance gains are realized by increasing the number of backends. If the size of the database and the size of the responses to the transactions remain constant, then MBDS produces a reciprocal decrease in the response times for the user transactions when the number of backends is increased. On the other hand, if the number of backends is increased proportionally with the increase in databases and responses, then MBDS produces invariant response times for the same transactions. A more detailed discussion of MBDS can be found in [Ref. 8].

E. THESIS OVERVIEW

The organization of our thesis is as follows: In Chapter 2, we discuss the software engineering aspects of our implementation. This includes a discussion of our design approach as well as a review of the global data structures used for the implementation. In Chapter 3, we outline the functionality of the language interface layer. In Chapter 4, we articulate the processes constituting the kernel mapping system. In Chapter 5, we provide an overview of the kernel controller. In Chapter 6, we describe the kernel formatting system. In Chapter 7, we conclude the thesis.

Appendix A covers the data structures diagrams for the shared and local data. The detailed specifications of the interface modules, i.e., LIL, KMS, KC, and KFS, are given in

Appendices B, C, D, and E, respectively. Appendix F is a users' manual for the system. The specifications of the source data language, SQL, and of the target data language, ABDL, can be found in either [Ref. 9] or [Ref. 3].

II. SOFTWARE ENGINEERING OF A LANGUAGE INTERFACE

In this chapter, we discuss the various software engineering aspects of developing a language interface. First, we describe our design goals. Second, we outline the design approach that we took to implement the interface. Included in this section are discussions of our implementation strategy, our software development techniques, and salient characteristics of the language interface software. Then, we provide a critique of our implementation. Fourth, we describe the data structures used in the interface. And finally, we provide an organizational description of the next four chapters.

A. DESIGN GOALS

We are motivated to implement an SQL interface for a MLDS using MBDS as the kernel database system, the attribute-based data model as the kernel data model, and ABDL as the kernel data language. It is important to note that we do not propose changes to the kernel database system or language. Instead, our implementation resides entirely in the host computer. All user transactions in SQL are processed in the SQL interface. MBDS continues to receive and process requests in the syntax and semantics of ABDL.

In addition, we intend to make our interface transparent to the user. For example, an employee in a corporate environment with previous experience with SQL could log into our system, issue an SQL request and receive result data in a relational format, i.e., a table. The employee requires no training in ABDL or MBDS procedures prior to utilizing the system.

B. AN APPROACH TO THE DESIGN

1. The Implementation Strategy

There is a number of different strategies we could have employed in the implementation of the SQL language interface. For example, there are the build-it-twice full-prototype approach, the level-by-level top-down approach, the incremental development approach, and the advancemanship approach [Ref. 10: pp. 41-46]. We have predicated our choice on minimizing the "software-crisis" as explained by Boehm [Ref. 10: pp. 14-31].

The strategy we have decided upon is the level-by-level top-down approach. Our choice is based on, first, a time constraint. The interface has to be developed within a specified time, specifically, by the time we graduate. And second, this approach lends itself to the natural evolution of the interface. The system is initially thought of as a "black box" (see Figure 1) that accepts SQL transactions and then returns the appropriate

results. The "black box" is then decomposed into its four modules (i.e., LIL, KMS, KC, and KFS). These modules, in turn, are further decomposed into the necessary functions and procedures to accomplish the appropriate tasks.

2. Techniques for Software Development

In order to achieve our design goals, it is important to employ effective software engineering techniques during all phases of the software development life-cycle. These phases, as defined by Ledthrum [Ref. 11: p. 27], are as follows:

(1) Requirements Specification - This phase involves stating the purpose of the software: what is to be done, not how it is to be done.

(2) Design - During this phase an algorithm is devised to carry out the specification produced in the previous phase. That is, how to implement the system which is specified during this phase.

(3) Coding - During this phase the design is translated into a programming language.

(4) Validation - During this phase it is ensured that the developed system functions as originally intended. That is, it is validated that the system actually performs what it is supposed to do.

The first phase of the life-cycle has already been performed. The research done by Demurjian and Hsiao [Ref. 1] has described the motivation, goals, and structure of the MLDS. The research conducted by Macy [Ref. 2] and Rollins [Ref. 3] has extended this work to describe in detail the purpose of the SQL interface.

Hence, the requirements specification is derived from the above research.

We have developed the design of the system using the above specification. A Systems Specification Language (SSL) [Ref. 12] is used extensively during this phase. The SSL has permitted us to approach the design from a very high-level, abstract perspective by :

(1) enhancing communications among program team members,

(2) reducing dependence on any one individual, and

(3) producing complete and accurate documentation of the design.

Furthermore, the SSL has allowed us to make an easy transition from the design phase to the coding phase.

We have used the C programming language [Ref. 13] to translate the design into executable code. Initially, we were not conversant in the language. However, our background in Pascal and the simple syntax of C have made it easy for us to learn. The biggest advantage of using C is the programming environment that it resides (i.e., the UNIX operating system). This environment has permitted us to partition the SQL interface and then manage these parts in an effective and efficient manner. Perhaps, the only disadvantage with using C is the poor error diagnostics, having made debugging difficult. There is an on-line debugger available for use with C in UNIX for debugging. We have avoided this option and instead used

conditional compilation and diagnostic print statements to aid in the debugging process. To validate our system we have used a traditional testing technique, i.e., path testing [Ref. 14]. We have checked boundary cases such as the nested select and the single select. And we have tested those cases considered "normal". It is noteworthy to mention that testing, as we have done it, does not prove the system correct, but can only indicate the absence of problems with the cases that have been tested.

3. Characteristics of the Interface Software

In order for the SQL interface to be successful, we have realized that it must be well designed and well structured. Hence, we are cognizant of certain characteristics that the interface must possess. Specifically, it must be simple. In other words, it must be easy to read and comprehend. The C code we have written has this characteristic. For instance, we often write the code with extra lines to avoid shorthand notations available in C. These extra lines have made the difference between comprehensible code and cryptic notations.

The interface software also must be understandable. This must be true to the extent that a maintenance programmer, for example, can easily grasp the functionality of the interface and the relation between it

and the other pieces of the system. Our software possesses this characteristic and does not have any hidden side-effects that could pose problems months or years from now. As a matter of fact, we have intentionally minimized the interaction between procedures to alleviate this problem.

The interface must also be maintainable. This is important in light of the fact that almost 70% of all of the software life-cycle costs are incurred after the software becomes operational, i.e., in the maintenance phase. There are software engineering techniques we employed that have given the SQL interface this characteristic. For example, we require programmers to document changes to the interface code when the change is made. Hence, maintenance programmers have current documentation at all times. The problem of trying to figure out the functionality of a program with dated documentation is alleviated. We also required the programmers to update their SSL specification as the code is being changed. Thus, the SSL specification consistently corresponds to the actual code. In addition, the data structures are designed to be general. Thus, it is an easy task to modify or rectify these structures to meet the demands of an evolving system.

The research conducted by Demurjian and Hsiao [Ref. 1] provides a high-level specification of the MLDS.

The theses written by Macy [Ref. 2] and Rollins [Ref. 3] extend the above work and provide a more detailed specification of an SQL language interface. This thesis outlines the actual implementation of an SQL interface. The appendices provide the specification SSL for this implementation.

A final characteristic that an SQL interface should have is extensibility. A software product must be designed in a manner that permits the easy modification and addition of code. In this light, we have placed "stubs" in the correct locations of the KFS to permit the easy insertion of the code needed to handle multiple horizontal screens of output. In addition, we have designed our data structures in a manner that will permit subsequent programmers to easily extend them to handle not only multiple users, but also other language interfaces.

C. A CRITIQUE OF THE DESIGN

Our implementation of the SQL interface possesses all of the elements of a successful software product. As noted previously, it is simple, understandable, maintainable, and extensible. Our constant employment of modern software techniques have ensured the success.

However, there are two techniques that are especially worthy of critiques. The first of these is the use

of the SSL. Initially, we have felt that the implementation language may also serve as the language to specify program algorithms. However, in doing so, we have stifled our creativity. This is because we are concentrating not only on what the algorithm does, but also on what the constructs (data structures) of the algorithm are. The use of the SSL has permitted us to concentrate on the functionality of the algorithm without a heavy concentration on its particular constructs. This has allowed us to view the algorithm in a detached manner so that the most efficient implementation for the constructs can be used. Although we have initially felt that the development of the program with the SSL may be too time-consuming, our opinions are changed when we have realized the advantages of the SSL and the overall complexity of the SQL language interface.

The way in which the data structures are designed is the other noteworthy software engineering technique. Being relatively inexperienced programmers, we are inclined to use static structures. Hence, we have made extensive use of structures which are bound at compile time. We soon realize that in doing so, the computing resources (e.g., data space) of the system are being depleted quite rapidly. Therefore, it is necessary for us to design the data structures in a way that they can be managed in a dynamic fashion. Most of the data structures of the

SQL interface are linked lists. This design affords us the most convenient way to efficiently utilize the resources of the system. It is an easy task to use the C language's malloc (memory allocate) function to dynamically create the elements of a list as we have needed them. In addition, the free command is useful in releasing these same elements to be used again.

D. THE DATA STRUCTURE

The SQL language interface has been developed as a single user system that at some point will be updated to a multi-user system. Two different concepts of the data are used in the language interface

1. Data shared by all users.
2. Data specific to each user.

The reader must realize that the data structures used in our interface and described below have been deliberately made generic. Hence, these same structures support not only our SQL interface, but the other language interfaces as well, i.e., DL/I, CODASYL-DML, and Daplex.

1. Data Shared by All Users

The data structures that are shared by all users are the database schemas defined by the users thus far. In our case, these are relational schemas, consisting of relations and attributes. These are not only shared by all users, but also shared by the four modules of the

MLDS, i.e., LIL, KMS, KC, and KFS. Figure 3 depicts the first data structure used to maintain data. It is important to note that this structure is represented as union. Hence, it is generic in the sense that a user can utilize this structure to support SQL, DL/I, CODASYL-DML, or Daplex needs. However, we will concentrate only on the relational model. In this regard, the first field of this structure points to a record that contains information about a relational database. Figure 4 illustrates this record. The first field is just a character array containing the name of the relational database. The next field contains an integer value

```
union dbid_node
  {
    struct    rel_dbid_node    *rel;
    struct    hie_dbid_node    *hie;
    struct    net_dbid_node    *net;
    struct    ent_dbid_node    *ent;
  }
```

Figure 3. The dbid_node Data Structure.

```
struct rel_dbid_node
  {
    char                      name[DBNLength + 1];
    int                       num_rel;
    struct    rel_node        *first_rel;
    struct    rel_node        *curr_rel;
    struct    rel_dbid_node   *next_db;
  }
```

Figure 4. The rel_dbid_node Data Structure.

representing the number of relations in the database. The third and fourth fields are pointers to other records containing information about each relation in the database. Specifically, the third field points to the first relation in the database while the fourth field points to the current relation being accessed. The final field is just a pointer to the next relational database.

The record rel_node contains information about each relation in the database. (See Figure 5.) This structure is organized in much the same fashion that the rel_dbid_node is organized. The first field of the record holds the name of the relation. The next field contains the number of attributes in this relation. The third and fourth fields point to other records which contain data on the first and current attribute of this relation. And finally, the last field is a pointer to the next relation in this database.

```
struct rel_node
  {
    char                    name[RNLength + 1];
    int                     num_attr;
    struct    rattr_node    *first_attr;
    struct    rattr_node    *curr_attr;
    struct    rel_node      *next_rel;
  }
```

Figure 5. The rel_node Data Structure.

Figure 6 shows the structure of the final record type used to support the definition of the relational database schema. The first field is also an array, holding, in this case, the name of the attribute. The second field serves as a flag to indicate the attribute type. For instance, an attribute can either be an integer, a floating point number, or a string. The characters "i", "f", and "s" are used, respectively. The third field indicates the maximum length that a value of this attribute type may possibly have. For example, if this field is set to ten and the type of this attribute is a string, then the maximum number of characters that a value of this attribute type may have is ten. The fourth field is also a flag used to indicate whether or not this particular attribute is a key. The last attribute just points to the next attribute in this relation. The reader may refer to Appendices B through E to examine how these data structures are used in the SSL.

```
struct rattr_node
  {
    char                    name[ANLength + 1];
    char                    type;
    int                     length;
    int                     key_flag;
    struct    rattr_node    *next;
  }
```

Figure 6. The rattr_node Data Structure.

2. Data Specific to Each User

This category of data represents information needed to support each user's particular interface needs. The data structures used to accomplish this can be thought of as forming a hierarchy. At the root of this hierarchy is the record type user_info that maintains information on all of the current users of a particular language interface. (See Figure 7.) The user_info record holds the ID of the user, a union that describes a particular interface, and a pointer to the next user. The union field is of particular interest to us. As noted earlier, a union serves as a generic data structure. In this case, the union can hold the data for a user accessing either an SQL language interface, a DL/I LIL, a CODASYL-DML LIL, or a Daplex LIL. The li_info union is shown in Figure 8.

We are only interested in the data structures containing information for each user that pertains to the SQL language interface. This structure is referred to as

```
struct user_info
  {
    char                    uid[UIDLength + 1];
    union    li_info        li_type;
    struct   user_info      *next_user;
  }
```

Figure 7. The user_info Data Structure.

```
union li_info
  {
    struct    sql_info    sql;
    struct    dli_info    dli;
    struct    dml_info    dml;
    struct    dap_info    dap;
  }
```

Figure 8. The li_info Data Structure.

sql_info and is depicted in Figure 9. The first field of this structure, curr_db_info, is itself a record and contains currency information on the database being accessed by a user. The second field, file, is also a record. The file record contains the file descriptor and file identifier of a file of SQL transactions, i.e., either queries or creates. The next field, sql_tran, is also a record, and holds information that describes the SQL transactions to be processed. This includes the number of requests to be processed, the first request to be processed, and the

```
struct sql_info
  {
    struct    curr_db_info    curr_db;
    struct    file_info       file;
    struct    tran_info       sql_tran;
    int                       operation;
    struct    ddl_info        *ddl_files;
    struct    tran_info       *abdl_tran;
    union     kms_info        kms_data;
    union     kfs_info        kfs_data;
    union     kc_info         kc_data;
    int                       error;
  }
```

Figure 9. The sql_info Data Structure.

current request being processed. The fourth field of the sql_info record, operation, is a flag that indicates the operation to be performed. This can be either the loading of a new database or the execution of a request against an existing database. When this field represents the execution of a request, it is encoded with the ABDL request type to be executed. The next field, ddl_files, is a pointer to a structure describing the descriptor file and template file. These files contain information about the ABDL schema corresponding to the current relational database being processed, i.e. the ABDL schema information for a newly defined relational database. The sixth field, abdl_tran, is a pointer to a record that describes the ABDL equivalents to the transactions written in SQL i.e., the translated SQL requests. Specifically, this is the first ABDL request, the current ABDL request, and the number of ABDL requests to be processed. This data is provided by the KMS and used by the KC. The next three fields, kms_data, kc_data, and kfs_data, are unions that contain information that is required by the KMS, KC, and KFS. These will be described in more detail in the next four chapters. The last field, error, is an integer value representing a specific error type.

E. THE ORGANIZATION OF THE NEXT FOUR CHAPTERS

The following four chapters are meant to provide the user with a more detailed analysis of the modules constituting the MLDS. Each chapter will begin with an overview of what each particular module does and how it relates to the other modules. The actual processes performed by each module are then discussed. This includes a description of the actual data structures used by the modules. Each chapter concludes with a discussion of module shortcomings.

III. THE LANGUAGE INTERFACE LAYER (LIL)

The LIL is the first module in the SQL mapping process, and is used to control the order in which the other modules are called. The LIL allows the user to input transactions from either a file or the terminal. A transaction can take the form of either creates for a new database or queries against an existing database. The mapping process takes place when the LIL sends a single transaction to the KMS. After the transaction has been received by the KMS, the KC is called to process the transaction. Control always returns to the LIL, where the user can close the session by exiting to the operating system.

The LIL is menu-driven. When the transactions are read from either a file or the terminal they are stored in a data structure called rel_req_info. If the transactions are creates they are sent to the KMS in sequential order. If the transactions are queries the user will be prompted by another menu to selectively pick an individual query to be processed. The menus provide an easy and efficient way to allow the user to see and select the methods in which to perform the mapping functions. Each menu is tied to its predecessor so that by exiting each menu the user is being

moved up the menu "tree". This allows the user to perform multiple tasks in one session.

A. THE LIL PROCESS

In this section we discuss the processes and actions performed by the LIL. These processes are presented in the order in which they are encountered during a typical session. The data structures used heavily by the LIL are discussed first.

1. Important Data Structures

The LIL uses two data structures to store the user's transactions and to control which transaction is to be sent to the KMS. It is important to note here that these data structures are shared by both the LIL and the KMS.

The first structure is named tran_info and is shown in Figure 10. The first field of this record, first_req, contains the address of the first transaction of the transaction list that was read from a file or the terminal. The second field, curr_req, contains the

```
struct  tran_info
   {
     struct  rel_req_info  *first_req;
     struct  rel_req_info  *curr_req;
     int                   no_req;
   }
```

Figure 10. The tran_info Data Structure.

address of the transaction currently being processed. The LIL sets this pointer to the transaction that the KMS will next process, and then calls KMS. The third field, no_req, contains the number of transactions currently in the transaction list. This number is used for loop control when printing the transaction list to the screen or when searching the list for a transaction to be executed.

The second data structure used by LIL is named rel_req_info. Each copy of this structure represents a user transaction and thus, is an element of the transaction list. The rel_req_info is given in Figure 11. The first field of this record, req, is a character string that contains the actual SQL transaction. The second field, in_req, is a pointer to a list of character arrays that each contain a single line of one transaction. After all lines of a transaction have been read, the line list is concatenated to form the actual transaction, req. The third field of this structure, req_len, contains the number of characters the transaction

```
struct  rel_req_info
   {
     char                    *req;
     struct  temp_str_info   *in_req;
     int                     req_len;
     struct  rel_req_info    *next_req;
   }
```

Figure 11. The rel_req_info Data Structure.

occupies. It is used to allocate the correct and minimal amount of memory space for the transaction. The last field, next_req, is a pointer to the next transaction structure rel_req_info, in the transaction list.

2. Procedures and Functions

The LIL makes use of a number of procedures and functions in order to create the transaction list, pass elements of the list to the KMS, and maintain the database schemas. Each of these procedures and functions will not be described in detail, but a general description of the LIL process will be discussed.

a. Initialization

The MLDS is designed to be able to accommodate multiple users, but is implemented to support only a single user. To facilitate the transition from a single-user system to a multiple user system, each user possesses his own copy of a user data structure when entering the system. This user data structure stores all of the relevant data that the user may need during their session. All four modules of the mapping process make use of this structure. The modules use many temporary storage variables in performing their tasks or for passing data between modules. The transactions, in user data language and mapped kernel data language form, are also stored in each user data structure. It is easy to see that the user structure provides consolidated, centralized

control for each user of the system. When a user logs onto the system, a user data structure is allocated and initialized. The user ID becomes the distinguishing feature to locate and identify different users. The user data structures for all users are stored on a linked list so that when a new user enters the system, their initialized user data structure is appended to the end of the list. In our current environment there is only a single element on the user list. In a future environment, when there are multiple users, we simply adopt the append operation mentioned above.

b. Creating the Transaction List

There are two operations the user can perform on the database schemas. A user can create a new database or process queries against an existing database. The first menu that is displayed prompts the user for which function to perform. Each function represents a separate procedure to handle the specific circumstances. This menu looks like the following:

```
Enter type of operation desired
     (l) - load a new database
     (p) - process old database
     (x) - return to the operating system

ACTION ----> _
```

For either choice (i.e., l or p), another menu is displayed to the user asking for the mode of input.

This input may come from a data file or interactively from the terminal. The generic menu looks like the following:

```
Enter mode of input desired
     (f) - read in a group of transactions from a file
     (t) - read in transactions from the terminal
     (x) - return to the previous menu

ACTION ----> _
```

Again, each mode of input picked corresponds to a different procedure to be performed. The transaction list is created by reading from the file or terminal looking for an end-of-transaction marker or an end-of-file marker. These flags tell the system when one transaction has ended and when the next transaction begins. When the list is being created, the pointers to access the list must be initialized. These pointers, first_req and curr_req, have been described earlier in the data structure section. Both pointers are set to the first transaction read, in other words, the head of the transaction list.

c. Accessing the Transaction List

Since the transaction list stores both creates and queries, two different access methods must be employed to send the two types of transactions to the KMS. We discuss the two methods separately. In both cases the KMS accesses a single transaction from the transaction list. It does this by reading the transaction

pointed to by the request pointer, curr_req, of the data structure, tran_info. (See Figure 10 again.) Therefore, it is the job of the LIL to set this pointer to the correct transaction before calling the KMS.

(1) Sending Creates to the KMS. When the user has specified the filename of creates (if the input is from a file) or typed in a set of creates (if the input is from the terminal), any further user intervention is not required. To produce a new database, it does not make sense to process only a single create out of a set of creates, since they all must be processed in a specific order. Therefore, the transaction list of creates is sent to the KMS in its entirety. A program loop traverses the transaction list, calling the KMS for each create in the list.

(2) Sending Queries to the KMS. In this case, after the user has specified his mode of input, he conducts an interactive session with the system. First, all queries are listed on the screen. As the queries are listed from the transaction list, a number is assigned to each query in ascending order, starting with the number one. The number is printed on the screen to the left of the first line of each query. Next, an access menu is displayed which looks like the following:

```
Pick the number or letter of the action desired
     (num) - execute one of the preceding queries
     (d)   - redisplay the list of queries
     (x)   - return to the previous menu

ACTION ----> _
```

Since queries are independent items, the order in which they are processed does not matter. The user has the choice of executing any number of queries. A loop causes the query listing and menu to be redisplayed after any query has been executed so that further choices may made.

d. Calling the KC

As mentioned before, the LIL acts as the control module for the entire system. When the KMS has completed its mapping process, the transformed transactions must be sent to the KC to interface to the kernel database system. For creates the KC is called after all creates on the transaction list have been sent to the KMS. The mapped creates reside in another list that the KC is going to access. Since queries are independent items, the user should wait for the results from one query before issuing another query. Therefore, after each query has been sent to the KMS, the KC is immediately called. The single mapped query resides on the same second list for the creates which the KC can access easily.

e. Wrapping-up

Before exiting the system, the user data structure described in Chapter II must be deallocated. The

memory occupied by the user data structure is freed up and returned to the operating system. Since all of the user structures reside in a list, the exiting user's node must be removed from the list.

B. SHORTCOMINGS

As used in this chapter, a transaction consists of a single request on a database. A transaction would normally be allowed to contain multiple requests, such as an insert, a query, and then a modify on some portion of a database. This feature is not incorporated into the present system, but it could be easily integrated at some later date.

IV. THE KERNEL MAPPING SYSTEM (KMS)

The KMS is the second module in the SQL mapping interface and is called from the language interface layer (LIL) when the LIL has received SQL input requests from the user. The function of the KMS is to: (1) parse the request to validate the user's SQL syntax, and (2) translate, or map, the request to an equivalent ABDL request. Once an appropriate ABDL request, or set of requests, has been formed, it is made available to the kernel controller (KC) which then processes the request for execution by MBDS. The KC is to be discussed in Chapter V.

A. AN OVERVIEW OF THE MAPPING PROCESS

From the description of the KMS functions above we immediately see the requirement for a parser as a part of the KMS. This parser validates the SQL syntax of the input request. It is the driving force behind the entire mapping system.

1. The KMS Parser / Translator

The KMS parser has been constructed by utilizing Yet-Another-Compiler Compiler (YACC) [Ref. 15]. YACC is a program generator designed for syntactic processing of token input streams. Given a specification of the input language structure (a set of grammar rules), the user's code to be

invoked when such structures are recognized, and a low-level input routine, YACC generates a program that syntactically recognizes the input language and allows invocation of the user's code throughout this recognition process. The class of specifications accepted is a very general one: LALR(1) grammars. It is important to note that the user's code we speak of here is our mapping code that is going to perform the SQL-to-ABDL translation. As the low-level input routine, we have utilized a Lexical Analyzer Generator (LEX) [Ref. 16]. LEX is a program generator designed for lexical processing of input character streams. Given a regular-expression description of the input strings, LEX generates a program that partitions the input stream into tokens and communicates these tokens to the parser.

The parser produced by YACC consists of a finite-state automaton with a stack and performs a top-down parse, with left-to-right scan and one token look-ahead. Control of the parser begins initially with the highest-level grammar rule. Control descends through the grammar hierarchy, calling lower and lower-level grammar rules which search for appropriate tokens in the input. As the appropriate tokens are recognized, some portions of the mapping code may be invoked directly. In other cases, these tokens are propagated back up the grammar hierarchy until a higher-level rule has been satisfied, at which time further translation is accomplished. When all of the necessary

lower-level grammar rules have been satisfied and control has ascended to the highest-level rule, the parsing and translation processes, and, therefore, the mapping process, is complete. In Section B, we give an illustrative example of these processes.

2. The KMS Data Structures

The KMS utilizes, for the most part, just four structures defined in the interface. It, naturally, requires access to the SQL input request and ABDL output request structures discussed in Chapter II, the rel_req_info and ab_req_info structures, respectively. However, the four data structures to be discussed here are only those unique to the KMS.

The first of these, shown in Figure 12, is a record that contains information accumulated by the KMS during the grammar-driven parse that is not of immediate use. This record allows the information to be saved until a point in the parsing process where it can be utilized in the appropriate portion of the translation process. The first field in this record, first_tgt, is a pointer to the head of a list of attribute names. These are the attribute names specified by the user request to retrieve information from, or insert information into, the database. This list is only utilized during SELECT or INSERT operations. The second field, templates, is also a record and holds the relation name(s) referenced in the user query. During join

```
struct  rel_kms_info
  {
  struct  target_list_info  *first_tgt;
  struct  templates_info    templates;
  struct  insert_list_info  *first_val;
  char                      *temp-str;
  char                      *join_str;
  struct  rel_kms_info      *next_nest;
  }
```

Figure 12. The rel_kms_info Data Structure.

operations, two relation names may be kept in this record. The third field, first_val, is a pointer to the head of a list of values. These are the values that an INSERT request desires inserted into the database. The fourth field, temp_str, is a pointer to a variable-length character string. The character-string length is a function of the input request length, and is allocated, when required, to accumulate intermediate translation results while parsing the WHERE boolean-clause of a user request. The fifth field, join_str, is also a pointer to a variable length character string. The character-string length is again a function of the input request length, and it is allocated to accumulate the translation for the second ABDL RETRIEVE request that is generated in response to a join operation. The sixth field, next_nest, is a pointer to another record of the same type. The next_nest field is used only during the translation of a nested SELECT statement, in which case

we require a list of rel_kms_info structures, one corresponding to each level of the nested SELECT query.

The remaining three data structures, shown in Figure 13, are records that are pointed to by the rel_kms_info record, as just described. Respectively, they represent a list of attribute names (the target list), a record of relation names (the templates), and a list of attribute values (the insert list). ANLength and RNLength are constants defining the maximum lengths of attribute and relation names, respectively. It should be noted that the value field in the insert_list_info record is a pointer to a variable length character string. Although attribute-names have a constant maximum length constraint, the length of

```
struct  target_list_info
  {
  char                        name[ANLength + 1];
  char                        tgt_rel[RNLength + 1];
  struct  target_list_info    *next_attr;
  }

struct  templates_info
  {
  char   name1[RNLength + 1];
  char   name2[RNLength + 1];
  }

struct  insert_list_info
  {
  char                        *value;
  struct  insert_list_info    *next_val;
  }
```

Figure 13. Additional KMS Data Structures.

attribute values in the database is limited only by the constraint placed on them by the user in the original database definition, and as such they may be of varying lengths.

At the end of the mapping process, before control is surrendered to the LIL, all data structures that are unique to KMS which have been allocated during the mapping process are returned to the free list.

B. FACILITIES PROVIDED BY THE IMPLEMENTATION

In this section, we discuss those SQL facilities that are provided by our implementation of the relational interface. We do not discuss the SQL to ABDL translation in detail. Rather, we provide an overview of the salient features of the KMS, accompanied by one illustrative example of the mapping process. User-issued requests may take two forms, SQL database definitions, or SQL database manipulations. Appendix C contains the design of our implementation, written in a system specification language.

1. Database Definitions

When the user informs the LIL that the user wishes to create a new database, the job of the KMS is to build a relational database schema that corresponds to the database definitions input by the user. The LIL initially allocates a new database identification node (rel_dbid_node shown in Figure 4) with the name of the new database, as input by the

user. The LIL then sends the KMS one database definition at a time, which takes the form of an SQL CREATE TABLE request as follows:

```
CREATE TABLE  table_name :

              field_name_1 ( type(length) [, NONULL] ),
              field_name_2 ( type(length) [, NONULL] ),
                  ...

              field_name_n ( type(length) [, NONULL] )
```

For each CREATE TABLE request, an additional relation node (rel_node shown in Figure 5) is added to the database schema under construction. It should be apparent from the preceding CREATE TABLE example that for each relation node, we must also add a list of attribute nodes (rattr_node shown in Figure 6) to the schema. The database identification node holds the number of relations in the schema and the database name, each relation node holds the number of attributes in that relation and the relation name, and each attribute node holds the attribute name, type, length, and primary key information.

When the LIL has forwarded all database definitions entered by the user, the result is a completed database schema, as shown in Figure 14. The relational database schema, when completed, serves two purposes. First, when creating a new database, it facilitates the construction of the MBDS template and descriptor files. Secondly, when

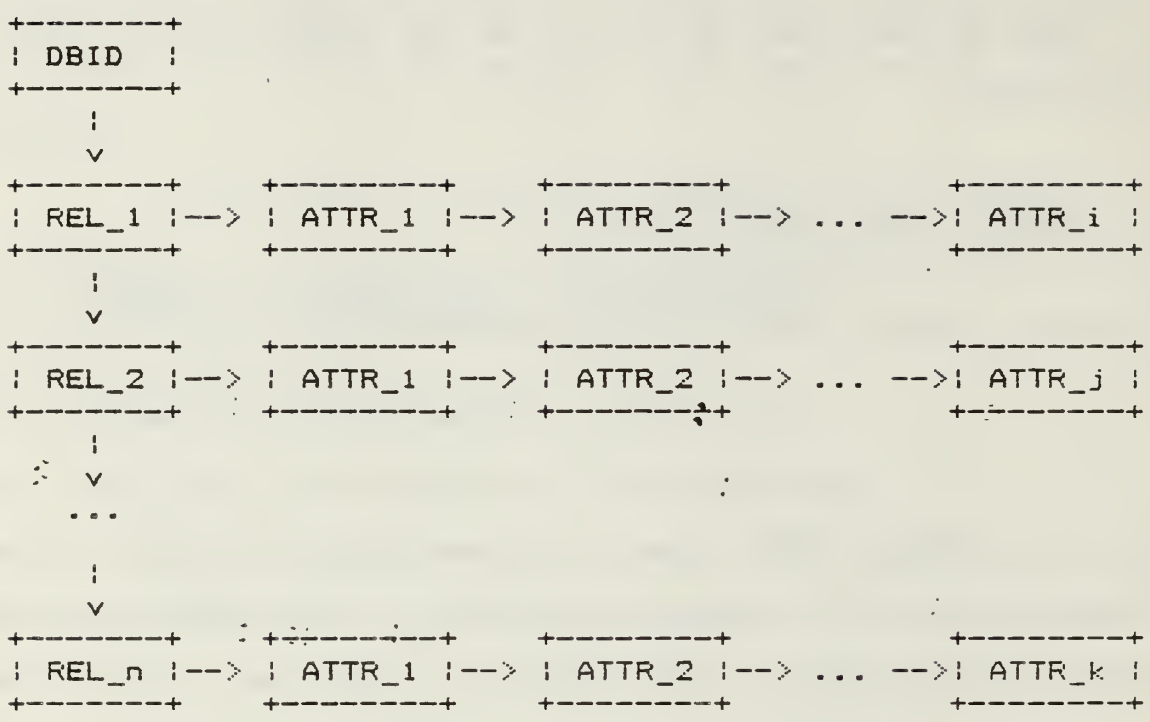


Figure 14. The Relational Database Schema.

processing requests against an existing database, it allows a validity check of the relation and attribute names. It also serves as a source of information for the type checking.

2. Database Manipulations

When the user wishes the LIL to process requests against an existing database, the job of the KMS is to map the user's SQL request to an equivalent ABDL request. Throughout this subsection, we only provide examples of the translated constructs of our implementation where they differ in some respect from those given in the work of Macy [Ref. 2] and Rollins [Ref. 3].

a. The SQL SELECT to the ABDL RETRIEVE

A simple SQL SELECT construct is mapped to a single ABDL RETRIEVE construct. A simple SELECT is characterized as a SELECT-FROM-WHERE block, in which access is limited to the information contained in a single relation of the database. The SELECT-clause may contain attribute names alone, or the aggregate functions (COUNT, SUM, AVG, MAX, and MIN) may be applied to any of the attributes where it makes sense to do so. The SELECT-clause may also contain an asterisk (*), which signifies that all attributes in the relation should be retrieved, in lieu of an exhaustive listing. As a final option, the attribute names may be prefixed with the relation name (rel_name.attr_name), even though only a single relation is being accessed. The FROM-clause contains this single relation name. The WHERE-clause may contain any number of predicates connected together by the boolean operators (AND and OR). Each predicate may utilize the six standard relational operators (=, /=, >, >=, <, and <=) to separate the attribute name and value, or the set membership operators (IN, NOT IN, /=ANY, <=ANY, <ANY, >ANY, >=ANY, <=ALL, <ALL, >ALL, >=ALL) may be used to separate the attribute name from an enumerated set of values. Finally, the SELECT-FROM-WHERE block may be optionally followed by either a GROUP BY-clause, or an ORDER BY-clause, whereby retrieved attributes may be either grouped or sorted.

A nested SQL SELECT construct is mapped to a series of ABDL RETRIEVE constructs. A nested SELECT is characterized as a SELECT-FROM-WHERE block, in which the WHERE-clause utilizes one of the set membership operators. In this instance, however, the operator is followed by another complete SELECT-FROM-WHERE block instead of an enumerated set of values. Such constructs can be nested to any depth. This allows multiple relations to be accessed, and their attribute-values compared, while the values returned to the user are taken from only a single relation. This is analogous to an implicit join operation. An example of such a query is as follows: Note that the parentheses are optional and need not be included.

```
SELECT  name, age
FROM    student
WHERE   name  IN
                ( SELECT  name
                  FROM    faculty )
```

This query would find the name and age of all students who are also a member of the faculty. It's ABDL counterparts would be as follows:

```
[ RETRIEVE  (TEMPLATE = FACULTY) (NAME) ]

[ RETRIEVE  ((TEMPLATE = STUDENT) and
             (NAME = ***********)) (NAME, AGE) ]
```

Notice that the first ABDL request corresponds to the last (or innermost) SQL request. This is because the innermost SQL request is the only one that represents a completely specified simple SELECT. The results of the first RETRIEVE are names which are used by the KC to fill in the place holders marked with asterisks in the second RETRIEVE (the number of asterisks equals the maximum length of the NAME attribute-value). From a single nested SELECT, the KMS generates a series of ABDL RETRIEVEs before relinquishing control to the KC, for subsequent execution of the ABDL requests.

A join SQL SELECT construct is mapped to a single ABDL RETRIEVE-COMMON construct. A join SELECT is characterized as a SELECT-FROM-WHERE block, in which the FROM-clause contains two relation names. We have already seen how the nested SELECT query specifies an implicit join. Here we are concerned with explicit joins, where multiple tables are accessed, and their attribute values compared, with the values returned to the user being taken from two different relations. In this instance, the SELECT-clause normally contains attribute-names that are prefixed with the appropriate relation name (rel_name.attr_name). This eliminates any ambiguity that might otherwise exist. The prefixed attribute-names are a required convention in the WHERE-clause. An example of such a query is as follows:

```
SELECT  student.name, faculty.name
FROM    student, faculty
WHERE   student.class = faculty.class
```

Assuming each class was only taught by one member of the faculty, this query would return a class roster for all members of the faculty. It's ABDL counterpart would be as follows:

```
[ RETRIEVE  (TEMPLATE = STUDENT) (NAME)
  COMMON    (CLASS = CLASS)
  RETRIEVE  (TEMPLATE = FACULTY) (NAME) ]
```

Notice the placement of the square brackets around the ABDL request. This represents a single ABDL request, and is forwarded to MBDS for execution as a single transaction. The use of prefixed attribute names in the SELECT-clause is not a necessity, providing that the attribute-names used are valid in both relations. Thus, the last SQL example may be entered as shown below to obtain the same results.

```
SELECT  name
FROM    student, faculty
WHERE   student.class = faculty.class
```

b. The SQL INSERT to the ABDL INSERT

The SQL INSERT construct is mapped to a single ABDL INSERT construct. If values are to be inserted for

each attribute in the relation, there is no requirement to list the attribute names. Only the attribute values need be listed; however, they must appear in the correct order (as listed in the schema which has been determined during the original database definition of the relation). If values are not inserted for each attribute in the relation, corresponding attribute names of those attribute values to be inserted must also be included in the request.

c. The SQL UPDATE to the ABDL UPDATE

The SQL UPDATE construct is mapped to a single ABDL UPDATE construct. ABDL does not provide a single-request construct which updates more than one attribute in a record. Thus, we only allow one predicate in the SET-clause of the SQL UPDATE query. However, the attribute value in this predicate may be a constant, or an arithmetic expression based on the original value of the attribute.

d. From the SQL DELETE to the ABDL DELETE: An Example

The SQL DELETE construct is mapped to a single ABDL DELETE construct. The SQL DELETE may have an optional WHERE-clause, so that all records for the particular relation may be deleted when the WHERE-clause is empty, or only those records satisfying a specific condition may be deleted when the WHERE-clause is included. In this subsection we will present an illustrative example of the mapping process for a simple SQL DELETE request. We begin

by showing the grammar for the delete-portion of the KMS. We then step through the grammar and show appropriate portions of our design in System Specification Language (SSL). The entire design is shown in Appendix C. The relevant grammar is as follows:

```
deletion :  DELETE  table_name  E;

table_name :  IDENTIFIER;

E :  empty
  |  WHERE  boolean;

empty :  ;

boolean :  ... ;
```

The source SQL request we will utilize for our example will be the following:

```
DELETE  student
```

It's ABDL translation will be as follows:

```
[ DELETE ( TEMPLATE = STUDENT) ]
```

To begin our discussion, let us first synchronize the reader. At the beginning of a mapping process, the parse descends the grammar hierarchy searching for appropriate tokens in the source that may satisfy one of

the grammar rules. Thus, the parser descends through the rules for SELECTs, INSERTs, etc. After finding no matching tokens for those rules, the parser eventually descends on the DELETE rules.

First, when the deletion rule is called, the DELETE-token will be recognized. In an attempt to satisfy the deletion rule, the table_name rule is then called. The table_name rule recognizes the IDENTIFIER-token, as the STUDENT-token (student converted to upper-case upon input). At this time, the table_name-rule is completely satisfied, and the following SSL is invoked:

```
table_name :  IDENTIFIER
                {
                if  (! creating)
                  if  (! valid_table('table_name') )
                    print ("Error - rel_name not valid")
                    perform yyerror()
                    return
                  end_if
                end_if
                };
```

If we are not creating a new database (as in this case), a call is made to the valid_table() function, which checks the validity of the IDENTIFIER table_name in the relational database schema. If STUDENT is not a valid relation name, then an error message is printed, and an error routine is called. Then we simply return from the mapping process. If STUDENT is a valid relation name, there is no code here for

translation; however, control returns to the rule that called the table_name rule (i.e., the deletion rule).

Next, even though the deletion rule is not completely satisfied, we need to perform some translation. The following SSL is invoked, before the call is made to the E-rule:

```
deletion:  DELETE  table_name
             {
             copy  "[ DELETE ( " to abdl_string
             copy  'table_name'  to templates
             }
           E;
```

The abdl_string begins to be built, as we initially copy

"[DELETE ("

into the abdl_string. The value of the table_name (STUDENT) is then copied to the templates data structure, because, at this point, we are not certain that it is of immediate use. The reader should note the trailing blank that we placed in the abdl_string. Without going into great detail, which is beyond the scope of this example, it suffices to say that this blank is for an additional left parenthesis that we may later determine to be required at the beginning of the ABDL request, i.e., when OR is used to connect WHERE-clause predicates.

The next step in the parse is for the deletion rule to call the E-rule. The E-rule recognizes the empty rule, because the source is now void of additional tokens. The E-rule is now completely satisfied and the following SSL is invoked:

```
E :  empty
       {
       delete_all = TRUE
       }
   |  WHERE  boolean;
```

This sets the delete_all boolean variable equal to true. Control now reverts to the deletion rule, which is of course completely satisfied. Thus, the following SSL is invoked:

```
deletion :  DELETE  table_name
            E
             {
             if  (delete_all)
               concat  "TEMPLATE = 'table_name'"
                                          to abdl_string
             end_if
             concat  ")"  to abdl_string
             };
```

Since we know that the delete_all variable has previously been set to true, we now concatenate

"TEMPLATE = STUDENT"

to the abdl_string. Finally, we concatenate the trailing

right parenthesis to the abdl_string. The trailing right bracket (]) is concatenated to the abdl_string after recognition of a higher-level grammar rule (one that called the deletion-rule), and the mapping process is now complete.

Let us continue with an extension of this example. Had the original SQL source request included a WHERE boolean-clause, such as the following, what would have happened?

```
DELETE  student
WHERE   name = 'Jones'
```

Its ABDL equivalent is as follows:

```
[ DELETE ( (TEMPLATE = STUDENT) and (NAME = Jones)) ]
```

In this instance, when the E rule is called, the WHERE-token would have been recognized, and thus the boolean rule would have been called. The boolean rule would have called other rules and continued to read the remainder of the input (source) tokens. Before the boolean rule was called, the abdl_string contained the following:

```
"[ DELETE ( "
```

When control returns to the E-rule, from the boolean rule,

the abdl_string will contain the following:

"[DELETE ((TEMPLATE = STUDENT) and (NAME = Jones)"

Then control would revert from the E-rule to the deletion rule. But this time, since the delete_all variable is not set to true in the E-rule, the deletion rule merely completes this portion of the translation by concatenating another right parenthesis to the abdl_string shown above. Again, the trailing right bracket is added at a higher level, and the mapping process is complete.

C. FACILITIES NOT PROVIDED BY THE IMPLEMENTATION

Our original intent has been to demonstrate that the relational interface could indeed be developed and implemented. As a demonstration, there are some facilities that are not included in our implementation. Some of these facilities have more to do with providing a user-friendly environment, than with supporting a germane relational interface. For others, the programming time and effort required to incorporate them would be too costly for the benefits derived. However, this is not to imply that such facilities would not be useful. This section is devoted to describing the most prominent features of SQL that are not supported by the language interface.

1. Interfacing Users

In our relational interface, there is no concept of a user view. A view may be thought of as a virtual relation that has no existence in its own right, but is derived from one or more existing relations. Under our implementation, the logical database and the physical database are one in the same. Thus, our interface is limited to data definition language (DDL) and data manipulation language (DML) statements, and provides no data-control facilities such as the GRANT and REVOKE options. Also, all CREATE TABLE requests are considered PERMANENT and SHARED. As mentioned in Chapter II, our interface data structures are constructed to facilitate future use by multiple users. This would allow the view concept to be supported by incorporating the relational database schemas into the user_information structure (user_info shown in Figure 7). These schemas would be virtual and user-specific with respect to the entire list of database schemas that are still global.

2. Updating Multiple Attributes

ABDL does not provide a single-request construct which updates more than one attribute in a record. The work of Rollins [Ref. 3: pp. 25-27] has showed that the SQL UPDATE may be translated into multiple ABDL requests. As a result, it may be necessary to generate either several independent ABDL UPDATEs, a transaction of ABDL UPDATEs (specifying the order in which a series of requests must be

processed), or an ABDL RETRIEVE, DELETE, and INSERT sequence, to accomplish the requested update of multiple attributes. We have felt the programming effort involved to provide such a facility, although not complex, is time-consuming.

3. Retrieving Qualified Groups

ABDL provides an option whereby retrieved attributes may be sorted (the by-attribute_name option). SQL provides a further option whereby those records not satisfying a specified condition can then be eliminated (the HAVING-condition option). ABDL does not provide a facility for checking this specified condition. It could have been implemented in the KC; however, we have felt the programming effort is too great for the benefits derived.

4. Retrieving Computed Values

This option supports the inclusion of arithmetic expressions involving attribute names in the SELECT-clause of SQL requests. An example of this option is as follows:

```
SELECT  name, weight * 454
FROM    student
```

This query would retrieve the name and weight of all students. However, the value of the attribute weight would be returned to the user in grams (found in the student relation in pounds). ABDL does not support the retrieval of

computed values; however, this could easily have been implemented in the KFS module. We have chosen not to implement, since it does not represent a feature of SQL that is inherently relational.

5. Eliminating Duplicates

The results of a SELECT query may contain duplicates. The elimination of duplicates is normally a high-cost operation and often unwarranted. We do not provide such an option. SQL supports the elimination of duplicates through the use of the UNIQUE operator in the SELECT-clause. Thus, our implementation does not support the SQL UNIQUE operator.

6. Retrieval Using UNION

The work of Rollins [Ref. 3: pp 82-83] has described the use of the SQL UNION operator in a query comprised of multiple SELECT constructs. Each SELECT construct translated to an equivalent ABDL RETRIEVE construct, and all are then processed by MBDS concurrently. Rollins has assumed the capability to eliminate duplicates in the interface. In as much as such a facility is not provided by our implementation, we are not supporting the SQL UNION operator.

V. THE KERNEL CONTROLLER

The kernel controller (KC) is the third module in the SQL language interface and is called by the language interface layer (LIL) when a new database is being created or when an existing database is being manipulated. In either situation the LIL first calls the kernel mapping system (KMS) which performs the necessary SQL to ABDL translations. Then the KC is called to perform the task of controlling the submission of the ABDL transaction(s) to the multi-backend database system (MBDS) for processing. If the transaction involves creating a new database or inserting, deleting or updating information in an existing database, control is returned to the LIL after MBDS processes the transaction. If the transaction involves a retrieval request, the KC sends the translated ABDL request to MBDS, receives the results back from MBDS, loads the results into a buffer and calls the kernel formatting system (KFS) to format the results a buffer at a time. After the last buffer is processed by the KFS the resulting table is displayed and control then returns to the LIL.

One situation worth noting is the processing of an SQL nested-select request. An n-level SQL nested-select is mapped to n corresponding ABDL retrieve requests.

Only the first ABDL retrieve (which corresponds to the innermost select) is a fully-formed request. All other ABDL retrieves which correspond to the outer-level selects are sent to the KC by the KMS as request templates. A request template is an ABDL retrieve request with one unspecified attribute value. The KC must use the results obtained from the previous ABDL retrieve request (i.e., attribute values) and the request template to build the next ABDL request, i.e., the KC substitutes the retrieved attribute values for the unspecified attribute value in the request template. The processing of nested selects is managed by the KC.

The procedures that make up the interface to the KDS (i.e., MBDS) are contained in the test interface (TI) [Ref. 8]. To fully integrate the KC with the KDS (i.e., MBDS), the KC calls procedures which are defined in the TI. Due to upcoming hardware changes in the MBDS, we decide not to test the KC on-line with the TI. Our solution to this problem is to design the system exactly as if it is interfacing with the TI. However, for each call to a TI procedure we create a procedure stub that performs the same function as the actual TI procedure. The reader should realize that all interactions with the TI procedures described in the KC are actually made with these procedure stubs, rather than with the on-line TI procedures.

In this section we discuss the processes performed by

the KC. This discussion is in two parts. First we examine the data structures relevant to the KC, which is followed by an examination of the procedures and functions found in the KC. Appendix D contains the design of our KC implementation, written in a system specification language.

A. AN OVERVIEW OF THE KC DATA STRUCTURES

In this section we will review the data structures mentioned in chapter 2, focusing on those structures that are accessed and used by the KC. The first data structure that is important to the KC is the record type sql_info shown in Figure 15. The fields of sql_info contain all of the data structures relevant to the KC, but the KC only uses several of the fields. The first field of this record, curr_db, is a record which is used by the KC when

```
struct sql_info
  {
     struct    curr_db_info          curr_db;
     struct    file_info             file;
     struct    tran_info             sql_tran;
     int                             operation;
     struct    tran_info             *abdl_tran;
     int                             answer;
     union     kms_info              kms_data;
     union     kfs_info              kfs_data;
     union     kc_info               kc_data;
     int                             error;
     int                             subreq_stat;
  }
```

Figure 15. The sql_info Data Structure.

interacting with MBDS. When the KC sends an ABDL transaction to MBDS for execution, the current database name must be sent with the request. The current database name is stored in the curr_db record.

The next field of the sql_info record type which the KC uses is operation. This fourth field of sql_info contains an integer that tells the KC what type of operation is to be performed. There are six possible types of operations which correspond to the six operations supported by the KC. These operations are a database creation request, a retrieve request, a retrieve-common request, a delete request, an insert request and an update request.

The next field used by the KC is abdl_tran, which is a record of type tran_info, and is shown in Figure 16. The first two fields of tran_info are variant records which store information on ABDL requests. The ABDL requests are stored in a record of type ab_req_info, shown in Figure 17. Both first_req and curr_req initially contain the first ABDL request which is loaded into the data structure by the KMS.

```
struct tran_info
  {
     union     req_info                first_req;
     union     req_info                curr_req;
     int                               no_req;
  }
```

Figure 16. The tran_info Data Structure.

```
struct ab_req_info
  {
     char                              *req;
     int                               rel_op;
     struct    ab_rel_info             *next_req;
  }
```

Figure 17. The ab_req_info Data Structure.

The no_req tells the KC how many requests are in the linked list of the ab_req_info structure. There will normally only be one request in this list, unless an SQL nested select is being processed. In that case, the no_req will correspond to the number of levels that there are in the nested select. The first request will always be a fully-formed ABDL request, while any additional requests will be ABDL request templates. The requests or request templates are stored in the ab_req_info record. The *req is a pointer to a character string which contains either the request or the request template. The rel_op field informs the KC which type of relational operator is contained in the corresponding request or request template. The eleven possible operators are IN, NOT IN, ~=ANY, <=ANY, >=ANY, <ANY, >ANY, <=ALL, >=ALL, <ALL and >ALL. The *next_req is a pointer which directs the KC to the next ABDL request.

The next field of sql_info that the KC uses is kfs_data, which is a variant record into which the responses received from MBDS are stored. From this storage buffer the KFS extracts the data returned from the KDS to be

formated. This storage buffer is only used when we are processing SQL selects which have been mapped to ABDL retrieves. The results of the ABDL retrievals are loaded into the storage buffer. When the buffer is filled the KFS is called. The process of filling the buffer and calling KFS is repeated by the KC until all results from the retrieval have been processed.

The next field used by the KC from the sql_info record type is kc_data. This field is a variant record which contains the record type kc_rel_info. This record type holds all of the information that is unique to the KC. This data structure is shown in Figure 18. The field file_status is a flag used to indicate the status of the current and future result files. Two files are necessary to

```
struct kc_rel_info
  {
     int                                file_status;
     struct    max_info                 max;
     struct    min_info                 min;
     int                                num_values_ffile;
     int                                num_values_cfile;
     struct    nsel_res_info            files;
     char                               *unfin_ret;
     int                                beg_conj;
     int                                end_conj;
     int                                beg_asterisk;
     int                                end_asterisk;
     int                                req_len;
     int                                req_status;
     int                                curr_pos;
     int                                res_len;
  }
```

Figure 18. The kc_rel_info Data Structure.

handle SQL nested selects. The future-results file holds results from the current retrieve being processed by MBDS, while the current-results file holds the results from the previous retrieve request, which are used to build the current retrieve request. The records max_info and min_info are identical data structures. Both structures allow for the storage of a character which indicates the data type of the attribute-values, i.e., integers, floating point numbers or strings. Both structures also contain a variant record which is used for the storage of the respective maximum or minimum value encountered in the resultant records. The num_values_ffile and num_values_cfile indicate the number of values stored in the future or current results file, respectively. The record type of files is nsel_res_info. The file field contains two identical records of type file_info. This record stores a file name and a file descriptor used for file manipulation in the C programming language. One record is for the current-results file and the other is for the future-results file.

The *unfin_ret is a character array used to store the request template sent to the KC by the KMS. The request template that is stored in the first field of the ab_req_info record type is loaded into unfin_ret. This transfer of the request template is necessary so that the fully-formed ABDL request that is constructed by the KC can

be stored back into the *req field of ab_req_info. The beg_conj, end_conj, beg_asterisk and end_asterisk are integer fields which store the positions they describe in the request template, i.e., (unfin_ret). The conjunction is that portion of the request template which must be repeated as many times as necessary to hold the values returned from the previous inner-level request. The asterisks indicate where in that request the attribute-values must be placed. The field req_len holds the value of the maximum size in bytes of the fully-formed ABDL request that the KC builds and sends to the KDS. The req_len is calculated by the KC and is used for allocating storage for the fully-formed ABDL request which is constructed from the request template.

The req_status is a flag used to indicate whether we are processing the first request or subsequent requests. Curr_pos is an integer-valued variable that is used to indicate our current position in the current-request file and that marks which attribute-value is the next one to be inserted into the request being constructed. The res_len is the last field in the record of type kc_rel_info and is an integer-valued variable which contains the length of the response buffer returned by the MBDS. This value is used to indicate when we have completed our movement through the response buffer.

The final field used by the KC in the sql_info record

type is subreq_stat. This integer-valued variable is a flag used when the KC is handling a nested select. The flag is set to indicate either that the last subrequest is being processed or an intermediate subrequest is being processed.

B. KC PROCEDURES AND FUNCTIONS

The KC makes use of a number of different procedures and functions to manage the transmission of the translated SQL queries (i.e.,ABDL requests) to the KDS. Not all of these procedures and functions will be discussed in detail. Instead, we hope to provide the reader with an overview of how the KC controls the submission of the various types of ABDL transactions to MBDS.

1. The Kernel_Controller

The procedure Kernel_Controller is called whenever the LIL has an ABDL transaction for the KC to process. This procedure provides the master control over all other procedures used in the KC. The first portion of this procedure initializes global pointers that are used throughout the KC. The other portion of the procedure is a case statement which calls different procedures based upon the type of ABDL transaction that is being processed. If a new database is being created, the procedure load_tables is called. If the transaction is a retrieve-common request, an insert request, a delete request or

an update request, the procedure rest_request_handler is called. If the transaction is a retrieve request, then the procedure select_requests_handler is called. If the transaction is none of the above, there is an error. An error message is generated and control is returned to the LIL.

2. The Creation of a New Database

The creation of a new database is the least difficult transaction that the KC handles. The procedure load_tables is called by the KC and performs two functions. First, the test interface (TI) procedure dbl_template is called. This procedure is used to load the database-template file created by the KMS. Next the TI procedure dbl_dir_tbls is called. This procedure loads the database-descriptor file. These two files represent the attribute-based metadata that is loaded into the KDS, i.e., MBDS. After execution of these two procedures, load_tables returns control back to the kernel_controller which in turn returns control back to the LIL.

3. Insert, Delete, Update and Retrieve-Common Requests

Insert, delete, update and retrieve-common requests are all handled in a similar fashion. For any of these four types of requests, the KMS sends the translated ABDL request to the KC for processing. The main task of the KC for these four categories of requests is to send the ABDL request to the KDS (MBDS) for processing. The KC handles these types

of requests by calling the procedure rest_requests_handler which calls the procedure sql_execute. The procedure sql_execute controls the submission of ABDL requests to the KDS. To control the submission process the procedure sql_execute uses two TI procedures and the procedure sql_chk_responses_left. In general, the procedure sql_execute sends the ABDL request to the KDS, waits for the last response to be returned from the KDS and then takes action appropriate for the type of request submitted and the response received. For any of the request types sent to the KDS an error response might be received back. In this situation, an error message is sent to the user. If an error response was not received, then the ABDL request was correctly processed. For insert, delete and update requests, the user is sent a message informing him that the operation has been successfully executed. For a retrieve-common request, the results returned by the KDS are sent to the KFS for formatting. Control then returns upward through the various procedures until it reaches the LIL.

4. Retrieve Requests

ABDL retrieve requests are the other category of requests that the KC processes. The processing of retrieve requests is more complex than the other types of requests, since multiple retrieves (which correspond to SQL nested-selects) may need to be processed. The procedure select_requests_handler is called to process ABDL retrieve

requests. If a simple SQL select is being processed, then only one ABDL retrieve is generated by the KMS. If an SQL nested-select is being processed, then two or more ABDL retrieve requests are generated by the KMS. Only the first ABDL retrieve for an SQL nested-select is a complete ABDL retrieve. The remaining ABDL retrieves are actually ABDL request templates. An ABDL request template and the results of the previous retrieve are combined by the KC to build the fully-formed ABDL request. The procedure select_requests_handler manages both possible situations. First, the procedure sql_execute is called to process the initial fully-formed ABDL retrieve request. If this request is not an SQL nested-select, no other ABDL request templates remain. If ABDL request templates are left to process, then a loop is entered to process these retrieves. This loop is repeatedly executed until all ABDL request templates have been processed.

An overview of the activities controlled by this loop is necessary to understand how the KC handles the SQL nested-select. One of the initial steps in the loop is a call to the procedure swap_files. This procedure obtains the results generated by the previous ABDL request (which are stored in the future-results file by the procedure file_future_results) and puts them into the current-results file, where they are used to build the next ABDL retrieve. The number of values in the current-results file (which has

been determined when the values are loaded into the file) is obtained for use in the procedure. After some initialization steps are executed an inner loop is encountered. This inner loop controls the actual building and executing of the current ABDL request template which corresponds to one of the outer-level SQL selects.

The inner loop calls the procedure build_request to produce the next fully-formed ABDL retrieve. Control in this procedure is branched based upon which of the eleven possible SQL operators is in the current request. These eleven possible SQL operators result in four possible situations. For the operators <=ANY, <ANY, >=ALL and >ALL the procedure one_conjunction is called with the maximum value of the results in the current results file passed as a parameter. (The maximum and minimum values were calculated by the procedure file_future_results when the values were loaded into the future-results file.) For the operators >=ANY, >ANY, <=ALL and <ALL the procedure one_conjunction is also called, this time with the minimum value of the results in the current-results file passed as a parameter. For the IN and ~=ANY operators the procedure n_conjunction is called. For the NOT IN operator the procedure not_in_conjunction is called. These three conjunction procedures all produce one or more fully-formed ABDL retrieves using the request template. The inner loop then calls sql_execute to process the ABDL retrieve. The inner

loop concludes with some steps that prepare for the next retrieve. The inner loop repeats as long as there are values left in the current-results file and the procedure one_conjunction has not been called. The outer loop then sets up for the next ABDL retrieve and concludes. A more detailed example of how the procedures n_conjunction, not_in_conjunction, and one_conjunction work will be covered in the following two sections.

a. The N_conjunction Procedure

The procedure n_conjunction uses the ABDL request template and the values from the previous ABDL retrieve stored in the current-results file to build a fully-formed ABDL retrieve. The ABDL request template contains a portion of the ABDL request which we have labeled the conjunction. This conjunction portion of the request template is bounded by the first set of outermost parenthesis in those requests handled by the procedure n_conjunction. This conjunction contains a number of predicates that are "and'ed" together. A predicate is a triple consisting of an attribute name, followed by a relational operator (i.e., >, >=, ...) followed by an attribute value. Recall that in an earlier discussion we stated that the request template contains an unspecified attribute-value. In our current terminology, this means that a predicate in the conjunction of the request template has an unspecified attribute value. To mark this value

within the request template character string, we use a series of asterisks, where the number of asterisks corresponds to the maximum attribute-value length. The procedure n_conjunction uses the conjunction portion repeatedly with each conjunction having a different value from the results file inserted in place of the asterisks. The conjunctions are then "or'ed" together to form the fully-formed ABDL retrieve.

We have chosen not to allow an unlimited number of conjunctions to be joined together into one ABDL request. Rather we have created an upper limit on the maximum number of conjunctions that may be joined together into a single ABDL retrieve. We call this constant NUM_CONJ. Thus, assuming we have NUM_CONJ set to ten, only ten conjunctions can be linked together in one ABDL retrieve. This means only ten values from the current-results file can be loaded into the retrieve, one per conjunction. If there are more than ten results in the current-results file, then more than one ABDL retrieve must be built. This situation necessitates the inner loop discussed in the procedure select_requests_handler.

We now look at an example, to fully understand the operation of the procedure n_conjunction. We will build the outer ABDL retrieve for the nested select presented presented in Chapter IV. The SQL nested select is as follows:

```
SELECT name, age
FROM   student
WHERE  name  IN
                ( SELECT name
                  FROM   faculty )
```

This query would find the name and age of all students who are also a member of the faculty. The KMS maps the SQL nested-select to the following two ABDL retrieves.

```
[ RETRIEVE (TEMPLATE = FACULTY)  (NAME) ]

[ RETRIEVE ((TEMPLATE = STUDENT) and
            (NAME = ***********))  (NAME, AGE) ]
```

The first ABDL retrieve which corresponds to the innermost SQL request is executed by the procedure sql_execute. The results of this retrieve will be names of personnel on the faculty are stored in the current-results file. We assume that there are three names returned and that they are Demurjian, Mack and Kloepping.

The procedure n_conjunction marks several locations in the request template the first time it is called for a particular SQL request. The procedure stores the location of the beginning and the end of the conjunction and the location of the first and last asterisk which delineates the unspecified attribute-value. In the previous example the conjunction is as follows:

```
((TEMPLATE = STUDENT) and (NAME = ***********))
```

This conjunction is to be used three times to construct the fully-formed ABDL retrieve. The ABDL retrieve built by the procedure n_conjunction is shown in Figure 19.

If there had been more than NUM_CONJ names in the current-results file, then more than one fully-formed ABDL retrieve would have to be generated for the corresponding SQL select. The first NUM_CONJ names would have to be inserted into predicates that are "or'ed" together. Another ABDL retrieve would then be built using the next NUM_CONJ names from the current-results file. ABDL retrieves would continue to be built until all names in the current-results file have been exhausted.

b. The Procedures Not_in_conjunction and One_conjunction

The procedure not_in_conjunction operates in a similar fashion to the procedure n_conjunction. The major difference is that the conjunction portion of the request

```
[ RETRIEVE (((TEMPLATE = STUDENT) and (NAME = DEMURJIAN)) or
            ((TEMPLATE = STUDENT) and (NAME = MACK))       or
            ((TEMPLATE = STUDENT) and (NAME = KLOEPPING)))
                 (NAME, AGE) ]
```

Figure 19. The ABDL Retrieve Generated by the Procedure N_conjunction.

template is smaller and the conjunctions are "and'ed" together rather than "or'ed" together. Suppose that we replace the IN operator in the previous SQL nested-select query with the NOT IN operator. The resulting SQL query would find the name and ages of all students who are not members of the faculty. The first ABDL retrieve would be identical to the first ABDL retrieve in the last example. The second ABDL retrieve would now be as follows:

```
[ RETRIEVE ((TEMPLATE = STUDENT) and
            (NAME ~= ***********)  (NAME, AGE) ]
```

The conjunction portion for the procedure not_in_conjunction would be as follows:

```
((TEMPLATE = STUDENT) and (NAME ~= ***********))
```

The procedure not_in_conjunction inserts the names from the current-results file into this conjunction and "ands" the conjunctions together. The fully-formed ABDL retrieve is shown in Figure 20.

```
[ RETRIEVE ((TEMPLATE = STUDENT) and (NAME ~= DEMURJIAN)
            and (NAME ~= MACK) and (NAME ~= KLOEPPING))
                (NAME, AGE) ]
```

Figure 20. The ABDL Retrieve Generated by the Procedure Not_in_conjunction.

If there are more than NUM_CONJ names in the current-results file, then, as before, additional ABDL retrieves would be generated. The multiple ABDL retrieves to be generated are handled identically as they are in the procedure n_conjunction.

The procedure one_conjunction manages a simpler situation. For the procedure one_conjunction we are also sent an ABDL request template by the KMS. In these type situations all that must be done is to replace the asterisks with the minimum or maximum value that has been passed into the procedure as a parameter. Thus, the procedure one_conjunction simply removes the asterisks and inserts the passed in value in its place. It is only necessary for this single ABDL retrieve to be generated once. For example, suppose we are processing the following SQL nested-select.

```
SELECT name, age
FROM   student
WHERE  age <ALL
                ( SELECT age
                  FROM faculty )
```

This SQL query will retrieve the names and ages of all students that have an age less than all the faculty ages.

In other words, this request finds the names and ages of all students who have an age less than that of the youngest faculty member. The KMS maps the SQL nested-select to the following two ABDL retrieves.

```
[ RETRIEVE (TEMPLATE = FACULTY)  (AGE) ]

[ RETRIEVE ((TEMPLATE = STUDENT) and
            (AGE <= ***))  (NAME, AGE) ]
```

Assume the first retrieve results in the ages of 54, 43, 37 and 39 being returned. Since the SQL operator is <ALL, the minimum age is passed to the procedure one_conjunction. The fully-formed ABDL retrieve that the procedure one_conjunction generates is shown in Figure 21.

```
[ RETRIEVE (TEMPLATE = FACULTY)  (AGE) ]

[ RETRIEVE ((TEMPLATE = STUDENT) and
            (AGE <= ***))  (NAME, AGE) ]
```

Figure 21. The ABDL Retrieve Generated by the Procedure One_conjunction.

VI. THE KERNEL FORMATTING SYSTEM (KFS)

The KFS is the fourth module in the SQL language interface and is called from the kernel controller (KC) when the KC has obtained the final results from MBDS. The results are passed to the KFS in one or more character buffers, called response buffers. If there is more than one response buffer, the KC calls the KFS again for each buffer. The KFS manipulates the contents of these buffer(s) to create an image of an SQL results table. This table is formated in a file on each call to the KFS. Hence, this allows the user to view the results of his queries as if he is working with an SQL-type database system. The following example illustrates this process:

1. The user issues a query:

```
SELECT NAME,AGE
FROM EMPLOYEE
WHERE AGE < 30
```

2. The query is processed by all modules of the interface. Eventually, the KC receives the final results.

3. The KC calls the KFS for each response buffer.

4. The KFS uses the response buffer to create the output table. For illustrative purposes, suppose that the response buffer contains the following data:

```
NAME JOHN AGE 29 NAME STEVE AGE 26
```

5. The KFS displays the appropriate SQL output table:

EMPLOYEE	
NAME	AGE
JOHN	29
STEVE	26

It is important for the reader to note that the table actually consists of two parts. The first part is the table heading and column headings. In the example above this is the attribute called NAME followed by the attribute AGE. These are column headings. The table heading consists of the name of the relation, EMPLOYEE. The second part is instances of these attribute names or results, i.e., attribute values. In our example, JOHN and STEVE are results pertaining to the attribute NAME; while 29 and 26 are the results pertaining to the attribute AGE.

A. THE KFS PROCESS

In this section we discuss the processes that the KFS uses to create an SQL output table. We present these processes in the same sequence as they are performed by the KFS. We begin this discussion, however, with an overview of those data structures unique to the KFS. This overview can facilitate our understanding of the C code that constitutes this module.

1. Overview of the KFS Data Structures

The KFS utilizes, for the most part, just three of the structures defined in the language interface. The first of these, shown in Figure 22, is a record that contains information needed by the KFS to process the results. The first field in this record, response, contains the result from MBDS which is loaded by the KC just prior to calling the KFS. The second field, curr_pos, tells the KFS where it is in the response buffer. This helps the KFS maintain a correct orientation in the response buffer. The next field, res_len, indicates the length of the response buffer. This value is mostly used as a halting condition. For instance, the KFS continues to pull characters out of the buffer while some index is less than or equal to the res_len. The next field, form_data, is a record and contains information about the output table heading. This record will be discussed in the following

```
struct kfs_rel_info
  {
    char                          *response;
    int                           curr_pos;
    int                           res_len;
    struct   table_header_info    form_data;
    struct   file_info            o_file;
    int                           status;
    struct   rattr_node           *first_rel;
    struct   rattr_node           *sec_rel;
  }
```

Figure 22. The kfs_rel_info Data Structure.

paragraph. The fifth field, o_file, is also a record. The o_file record contains the file name and the file identifier of the file that the KFS is building the output table in. This is needed by the C language to open the output file for read, write, or append access. The next field, status, acts as a flag. If this is the first time the KFS is entered for a particular set of response buffers and, therefore, a particular user, then this field contains a value of FIRSTIME. This tells the KFS that it needs to initialize values and set various structures for subsequent processing. The status is changed after this is completed so that this initialization is not to be repeated for subsequent calls to the KFS for the same set of responses. The seventh field, first_rel, is a pointer to a list of attributes for the relation being currently processed. The data pertaining to this list can be considered the schema of the current relation. The specific data that is needed from the schema is the maximum size (in terms of the maximum attribute length) that the attribute named in this structure can possibly take on. This information is needed so that the correct column width for each attribute can be built into the output table. The final field is used for the same reasons discussed above, but is needed to implement the JOIN command.

The second structure the KFS uses extensively is also a record and is called table_header_info and is depicted in Figure 23. The purpose of this record is to provide information about the heading of the output table. The first field, table_width, is an integer value containing the width of the output table. This information tells us whether or not the table can fit within one horizontal screen width. This serves as the basis for some of our logic in the KFS and is discussed in detail in the next section. The next field, first_ent, points to another record that contains information about the first attribute name in the heading of the output table. The last field is the same as the previous one except that it points to the the current table_entry_info record that the KFS is now working with.

The third data structure, like the previous two, is also a record. This record maintains all of the information needed to correctly position attribute names in the heading of the output table. It also contains

```
struct table_header_info
  {
    int                          table_width;
    struct   table_entry_info    *first_ent;
    struct   table_entry_info    *curr_ent;
  }
```

Figure 23. The table_header_info Data Structure.

the information needed to correctly position the results under the appropriate headings. The first field, shown in Figure 24, is a character array containing the name of an attribute that is used as part of the output table heading. The second field is an integer value containing the length of the stored attribute name. This value is compared with the next field to determine the actual width of the column for this particular attribute. The next field, val_len, contains an integer value that is the maximum size a result of this attribute type can possibly take on. The fourth field, col_len, holds the maximum of the two previous fields and is the actual width of a column for a particular attribute in the output table. The last field, next, is just a pointer to the next record of this type. Using this field the KFS can move from one record to another record and create the correct heading until it hits a NULL record.

```
struct table_entry_info
  {
    char                          attr[ANlength + 1]
    int                           name_len;
    int                           val_len;
    int                           col_len;
    struct  table_entry_info      *next;
  }
```

Figure 24. The table_entry_info Data Structure.

2. KFS Procedures and Functions

The KFS makes use of a number of procedures and functions to create a finished SQL results table from a response buffer. We do not discuss all of these in detail. Rather, we do provide an overview of the more salient ones. We hope the reader may gain a more rounded appreciation of our effort from this discussion.

a. Initializing

Whenever the KFS is called, the initialization is required. This process basically involves setting values in the KFS data structures. For instance, the curr_pos is set to the first character in the response buffer. The initialization procedure also opens the output file in which the SQL table is to be created for the proper access. When the KFS is called for the first time, this file is opened for write access. Subsequent calls to the KFS for the same set of queries open this file for append access. This is done so that we do not overwrite data already in this file. The appropriate actions taken by the initialization routine is controlled by the value in the status. This value is updated by this routine after the first call.

b. Filling the Table Headings

The first time that the KFS is called the response buffer is scanned so that information may be gathered about the attribute names of the results. This

information is used by the KFS to create the header part of the output table.

This routine begins by reading the first attribute from the response buffer. The string length of this attribute is determined next; followed by a trace through the list of attributes in the schema of the current relation. The trace is completed when this attribute is found in the schema. At this point, the maximum size that a value of this attribute type may take on can be determined. This information is stored in val_len. The maximum of val_len and the string length of the attribute is then calculated and placed in col_len. This value represents the actual column width this attribute will have in the output table. The unwary reader may miss the importance of this step. It is easy to assume that the only value needed is val_len. However, let's assume there is an attribute called ZIP_CODE. The maximum number of characters this attribute may have is five digits. If we do not consider the string length of this attribute name, then the column size would be just five characters wide and ZIP_CODE would appear as ZIP_C in the heading.

This process of reading the next attribute is iterated until either it has cycled through a series of unique attribute names or it has processed all the attributes in the response buffer.

c. Creating the Table in the Output File

This part of the KFS can be considered the workhorse of the module. The previous two processes are instrumental in manipulating data structures and setting variable values so that this process may fulfill the intended mission of the KFS. Therefore, we discuss some of the issues we have struggled with while designing this part of the KFS. The two most important issues are:

1. How should the table appear to the user?
2. How should the table be stored internally, i.e., should it be in a file, a character array, or displayed immediately to the user?

Our problem has been that we have had no concrete examples of what an SQL table should look like. Should the headings be centered within the columns, with the results centered under these headings? We didn't know. We finally decided upon a convention that would facilitate programming. Hence, we left-justified both the headings and the results with blanks added at the end of each to insure proper spacing within the columns. As it turned out, this is also the way Date [Ref. 17: pp. 117-142] presents his examples.

The second issue has posed a problem. Our initial design has called for building the table in a character array. The only other alternative considered at this time has been to immediately put the table on the screen as results are being passed to the KFS. This idea

is dismissed, however, when we have realized the difficulties of trying to build a table on the fly. In a similar fashion, the idea of building the table in a character array is also dismissed. There is no way for us to predetermine the size of this array. We have thought that it is uneconomical to allocate a huge array to cover all possible table sizes. There is also the problem of moving around in the array. This indexing problem created a preponderance of C code.

Our only other alternative is to build the table in a file. This method has proved very easy to do. The operating system maintains position within the file, so, there is no indexing problem. In addition, there is more economical use of the computer's resources, since the file is only as big as necessary. Hence, we have opted to build the table in a file.

With these issues resolved, the implementation of this process has been straightforward. First, the headings are built in the output file. This is done only the first time the KFS is called. Next, the attribute values are pulled from the response buffer and placed left-justified under the corresponding attribute. Then, the process is iterated until the

response buffer is exhausted. Subsequent calls to the KFS for the same set of queries cause results to be appended to the table in the file.

d. Displaying the Table

The KFS displays the SQL table to the user when all response buffers have been processed. This occurs when a special signal is detected in the last buffer in conjunction with the setting of a status signifying the last sub_request of a nested select. An initial problem we have had with this process has been how to display a table more than twenty-four lines long. A unique procedure, patterned after the 'more' facility in UNIX [Ref. 18], is developed. This function, when first called, displays the first twenty-two lines of the SQL table and then prompts the user. The user can choose from a number of options. For instance, the user can have another screen-full of results displayed, or the user can display some number of lines less than twenty-two, or the user can even terminate the current menu of the language interface. Our intent is to make viewing the results as convenient as possible to the user.

e. Cleaning Up

Before leaving the KFS, the data structures used to create the SQL table are freed. This ensures that the resources are available to process other queries. Additionally, the status field is updated to FIRSTIME. This

places the KFS in the correct state to process subsequent queries correctly.

B. A LIMITATION OF THE KFS

Although we have tried to make the KFS as general as possible with regard to creating and displaying SQL tables, there is one facility we have deliberately neglected to incorporate. This is the ability to display tables with widths greater than eighty columns. Since our intent is to only show that the interface could indeed be developed, we have decided that the programming effort required to provide this facility is too costly for the benefits derived. However, this is not to imply that this facility is not useful or needed. As a matter of fact, we have intentionally designed the KFS for the easy insertion of this code when it is developed.

VII. CONCLUSION

In this thesis, we have presented the specification and implementation of a SQL language interface. This is one of four language interfaces that the multi-lingual database system will support. In other words, the multi-lingual database system will be able to execute transactions written in four well-known and important data languages, namely, SQL, DL/I, Daplex, and CODASYL. SQL is of course the well-known relational data language provided by, for example, IBM SQL/Data System. In our case, we support SQL transactions with our language interface by way of LIL, KC, KMS, and KFS in place of SQL/Data System. A related thesis by Benson and Wentz [Ref. 19] examines the specification and implementation of the DL/I language interface. Two other theses on CODASYL and Daplex respectively are under way. This work is part of ongoing research being conducted at the Laboratory of Database Systems Research, Naval Postgraduate School, Monterey, California.

The need to provide an alternative to the development of separate stand-alone database systems for specific data language models has been the motivation for this research. In this regard, we have first demonstrated the feasibility of a multi-lingual database system (MLDS) by showing how a

software SQL language interface can be constructed. Specific contributions of this thesis include the development of useful algorithms and the implementation of SQL operations such as: nested retrieval, join operations, retrieval of grouped attributes, and updating multiple fields. In addition, we have developed a LIL that is virtually reusable. With minor modifications the LIL can be used with the other language interfaces. As a matter of fact, it has been recently modified for the DL/I language interface. Our design of the generic data structures is also noteworthy. Because of our extensive utilization of unions (i.e., variant records), the other language interfaces can use our generic data structures. We have extended the work of Macy [Ref. 2] and Rollins [Ref. 3] by specifying and implementing the algorithms for the language interface. In addition, we have also provided a general organizational description of a MLDS.

A major goal has been to design a SQL-to-MBDS interface without requiring any change be made to MBDS or ABDL. Our implementation may be completely resident on a host computer or the controller. All SQL transactions are performed in the SQL interface. MBDS continues to receive and process transactions written in the unaltered syntax of ABDL. In addition, our implementation has not required any change to the syntax of SQL. The interface is

completely transparent to the SQL user as well as to the MBDL.

In retrospect, our level-by-level, top-down approach to the design of the interface has been a good choice. This implementation methodology has been the most familiar to us and proved to be relatively efficient in time. In addition, this approach permits follow-on programmers to easily maintain and modify (when necessary) the code. Subsequently, they will know exactly where we have stopped and where they should begin because we have included many of the lower-level stubs. Hence, it is an easy task of filling in these stubs with code.

We have shown that a SQL interface can be implemented as part of a MLDS. We have provided a software structure to facilitate this interface, and we have developed the actual code for implementation. The next step is to implement the other interfaces. When these are complete, the system needs to be tested as a whole to determine how efficient, effective, and responsive it is to users' needs. The results may be the impetus for a new direction in database system research and development.

APPENDIX A - SCHEMATIC OF THE DATA STRUCTURES

The purpose of this appendix is to present a pictorial of data structures used in the SQL language interface. Since the code used for the thesis was the C programming language, the diagram makes use of its constructs just as the code does. Groups of related items are known as structures in C, and it is easy to see from the diagram that the structures break down into more detailed, workable structures. There are two major parts of this appendix. In Figure 25 we present the relational database schema data structures that were discussed in Chapter 2. In Figure 26 we present the user data structures.

In the diagrams an arrow indicates that the field is a pointer to a structure. Each of the fields of such a structure is preceded by a small arrow to indicate that indeed a pointer from another structure is referencing the field. An example of this is the field si_ddl_files of the SQL_INFO structure in Figure 26 on page 113. The field si_ddl_files points to a structure of type DDL_INFO. This is especially useful when writing or tracing long paths through the user data structure.

On the other hand, bracket lines are used to indicate when a field of a structure is also a structure. The bracket lines are drawn from the "parent" field to the "child" structure. A period is placed in front of the bracketed structure's fields to indicate this fact. An

example of this is the si_sql_tran field of the SQL_INFO structure in Figure 26 on page 112. The field si_sql_tran is a structure of type TRAN_INFO. The bracket lines and the periods indicate this.

We note that the diagram has a few instances of UNIONS. A union is a construct that allows the user to connect different structure types, specified by the union structure, to a common structure, i.e., unions are also refered to as variant records. Since the multi-lingual database system is to support the mapping of multiple languages, many portions of the user structure will be the same for any language used. However, the union construct allows for the parts that must change between language interfaces so that the common data structures can be adapted to be useful to all of the language interfaces.

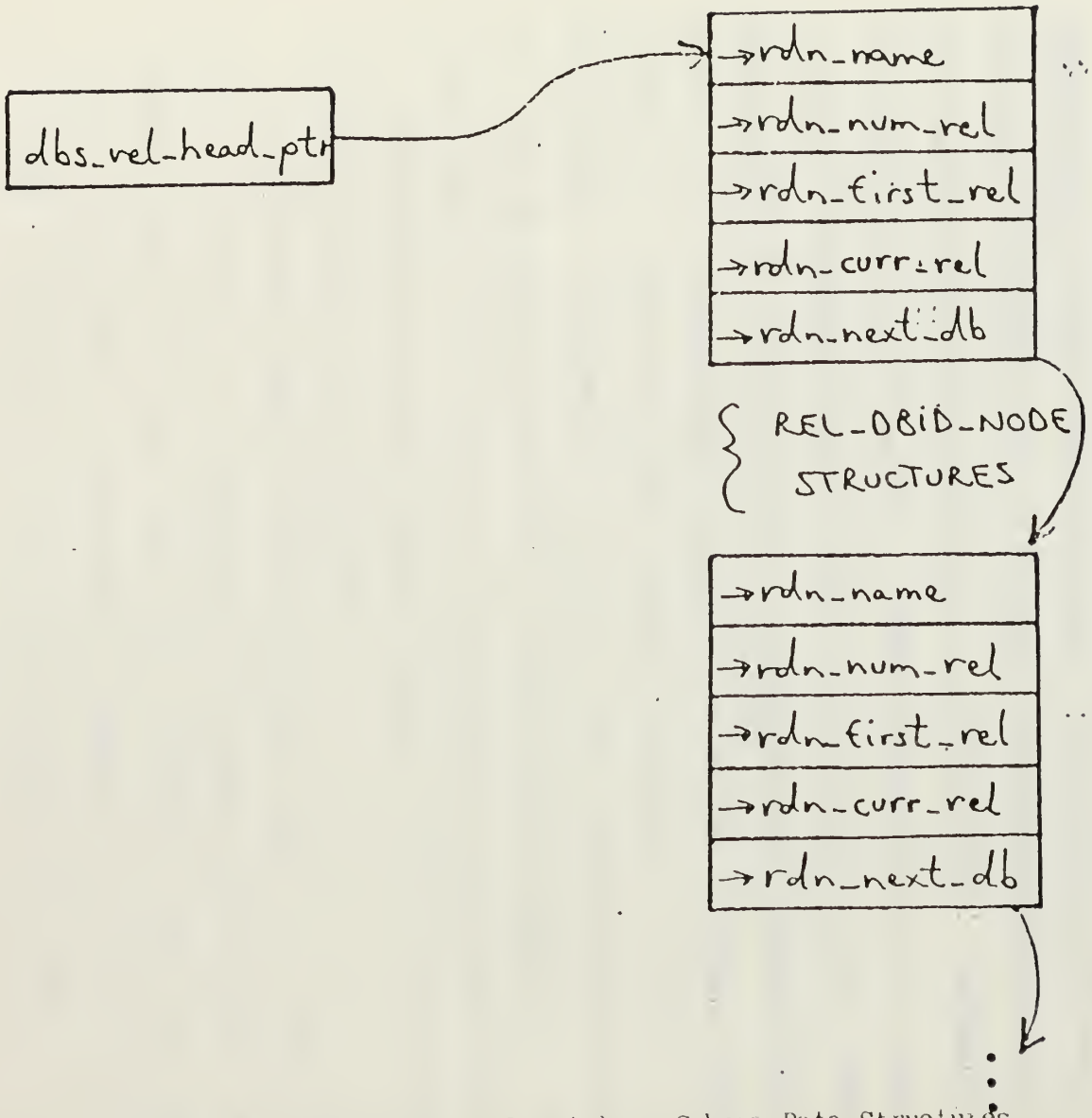


Figure 25. The Relational Database Schema Data Structures.

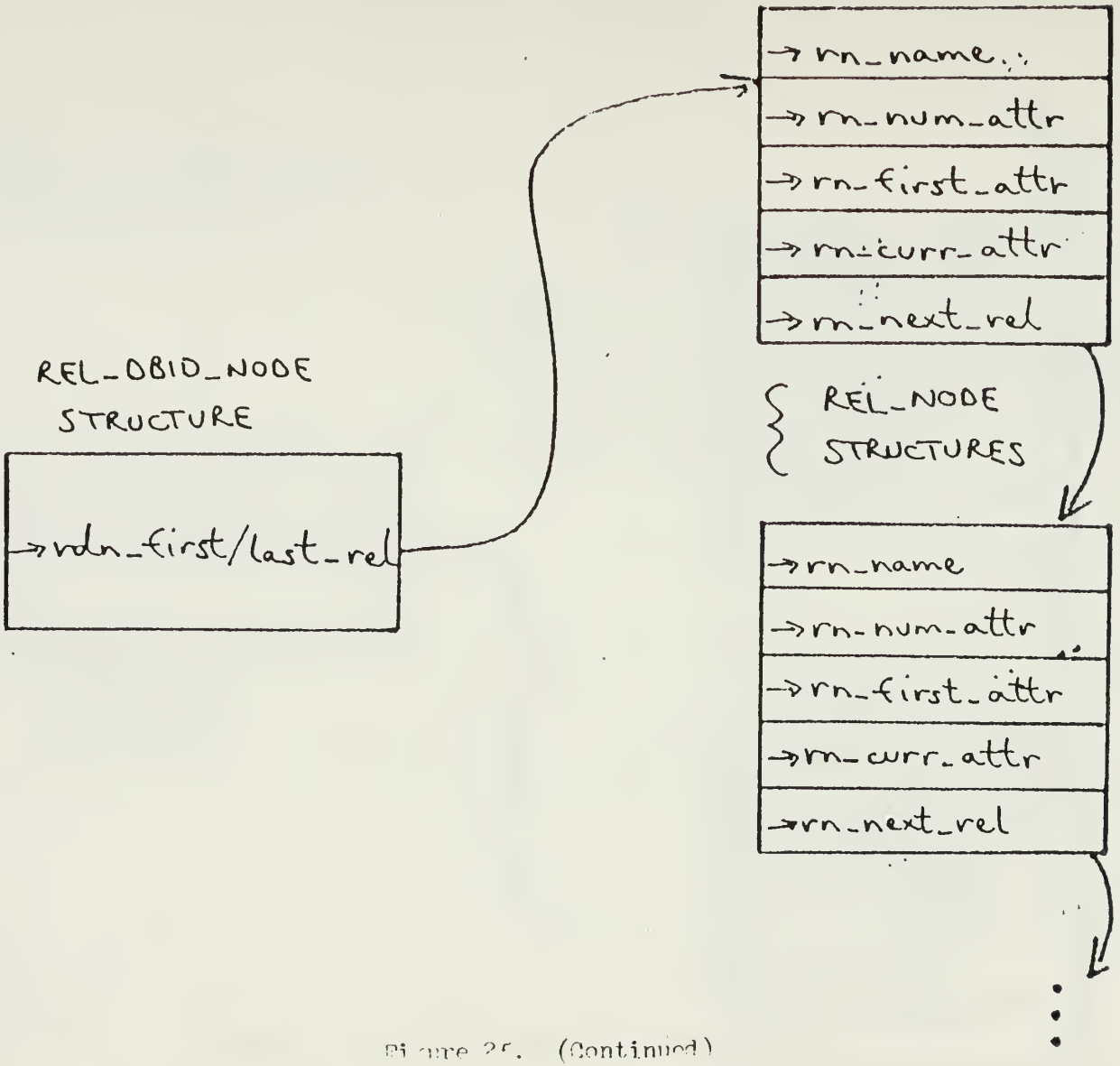


Figure 25. (Continued)

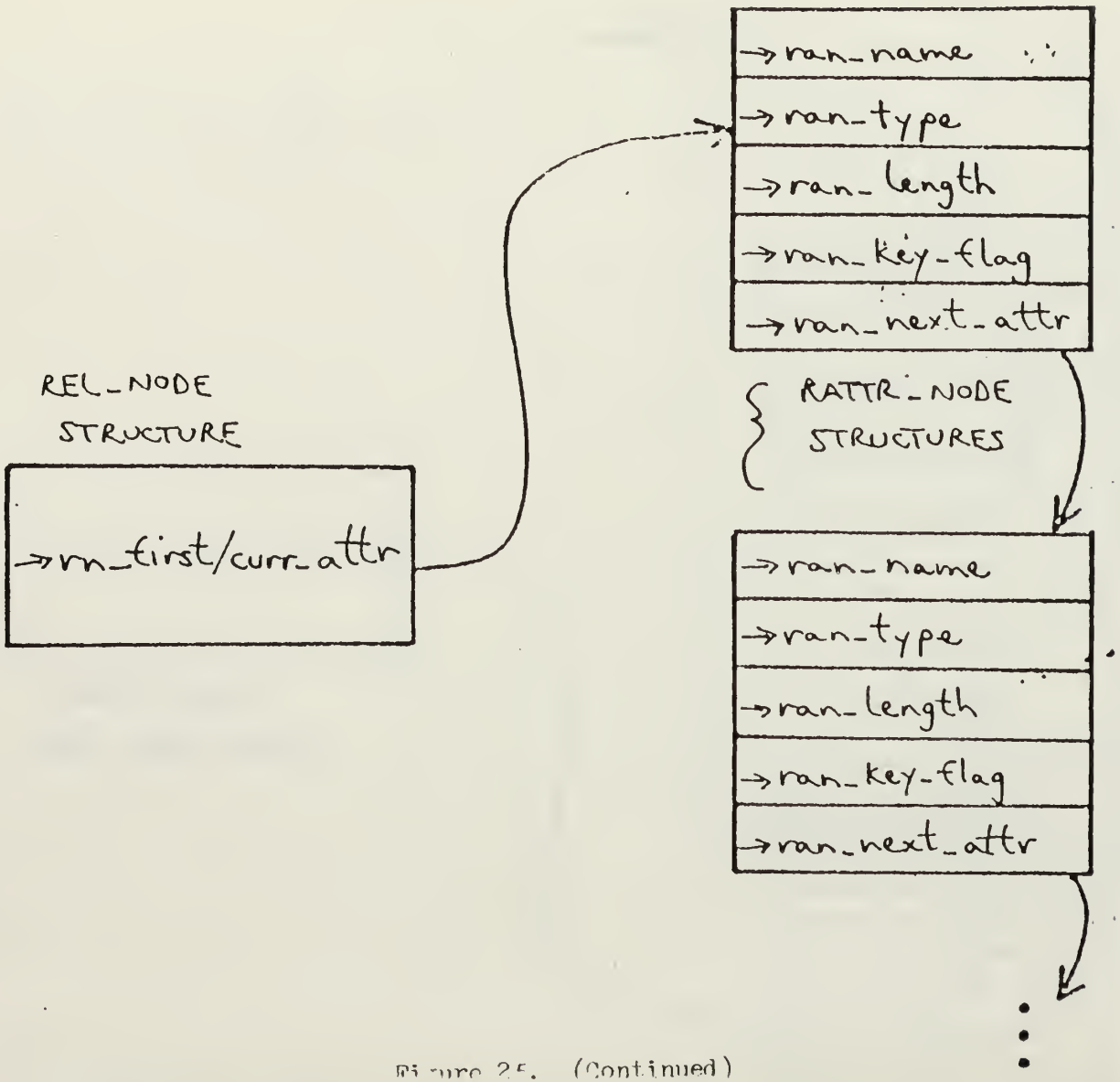


Figure 25. (Continued)

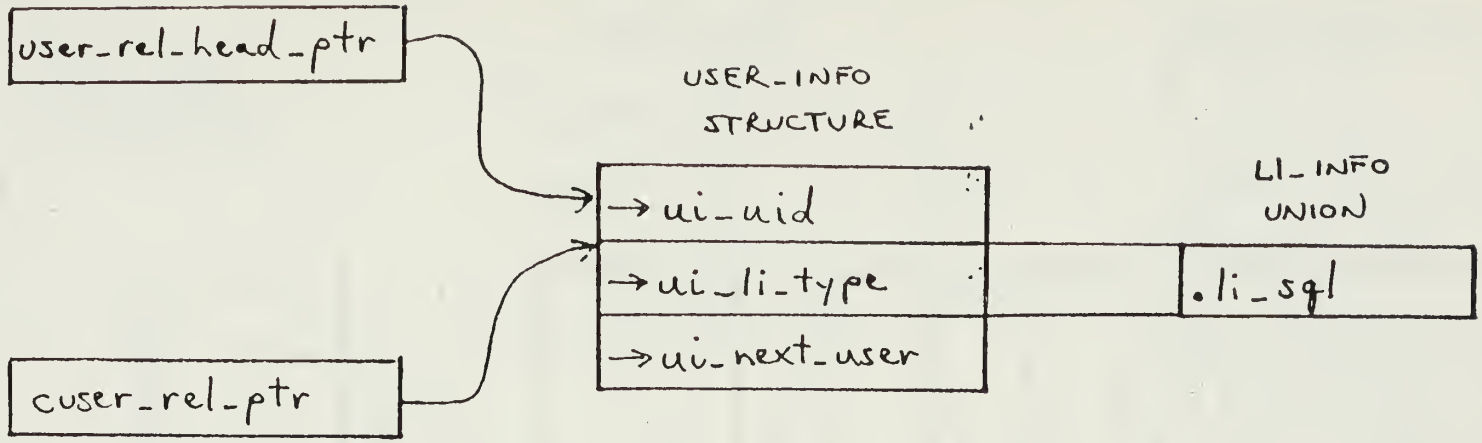


Figure 26. The User Data Structures.

SQL_INFO STRUCTURE

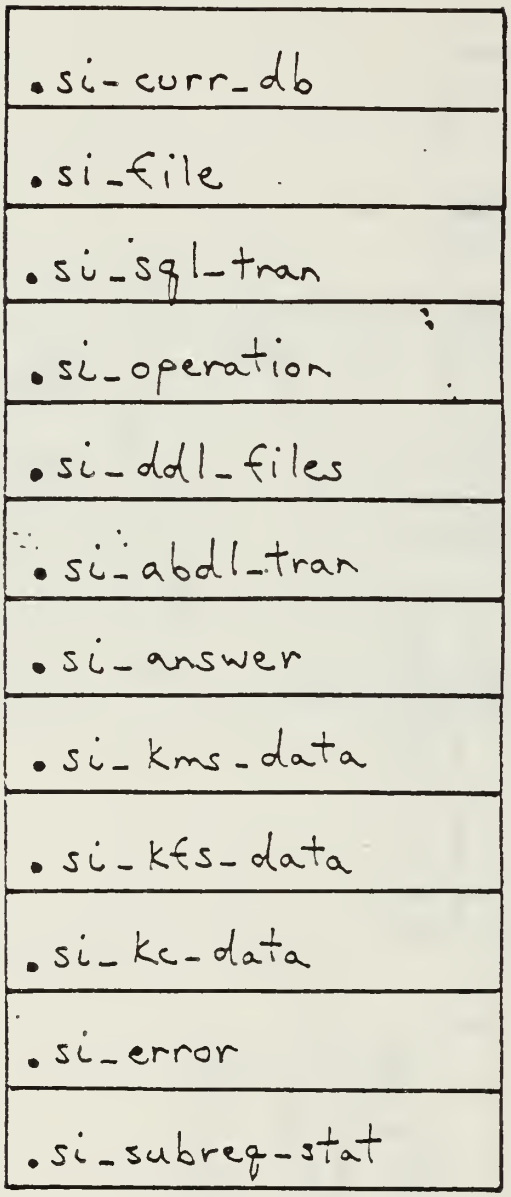


Figure 26. (Continued)

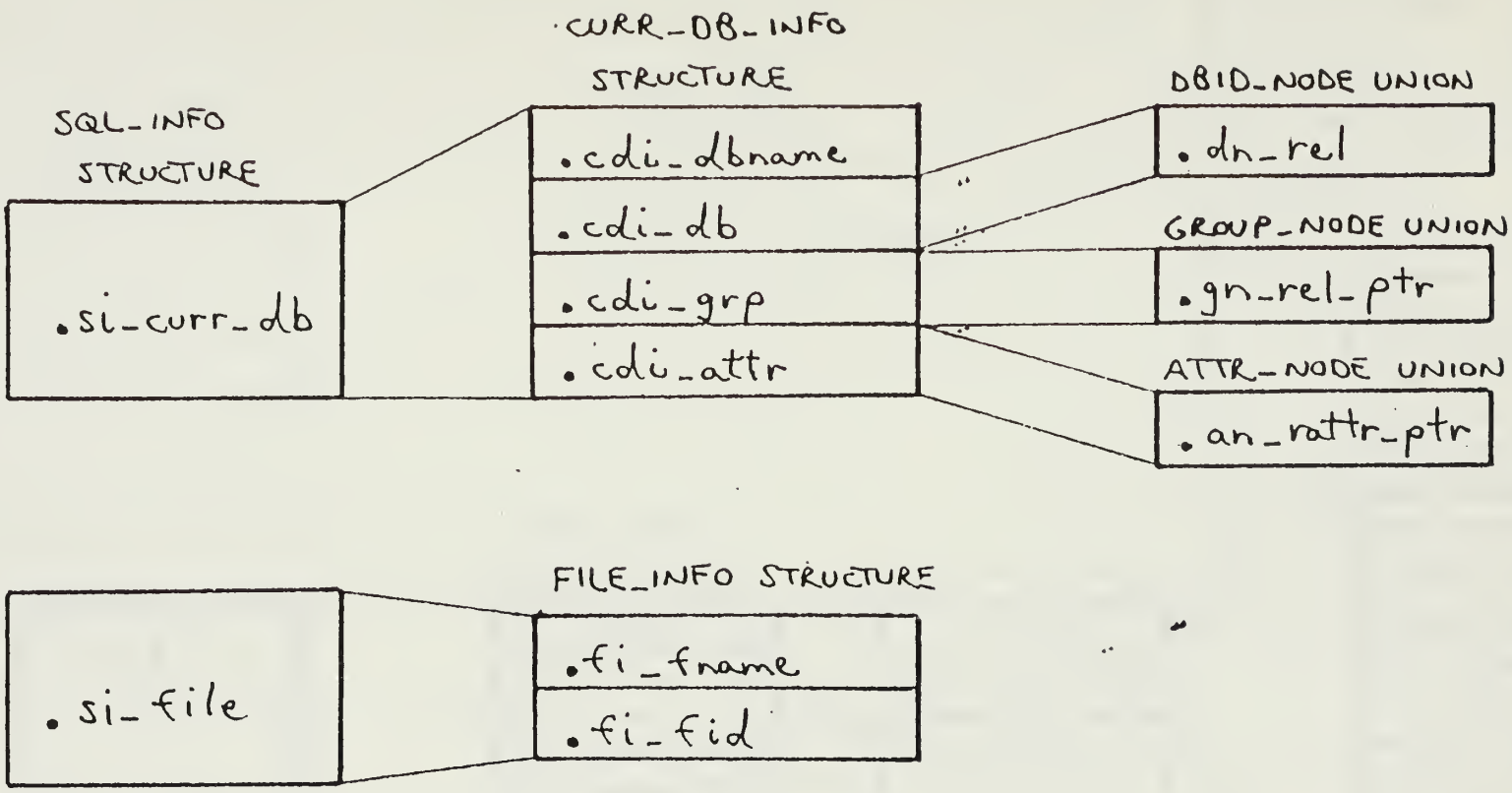


Figure 26. (Continued)

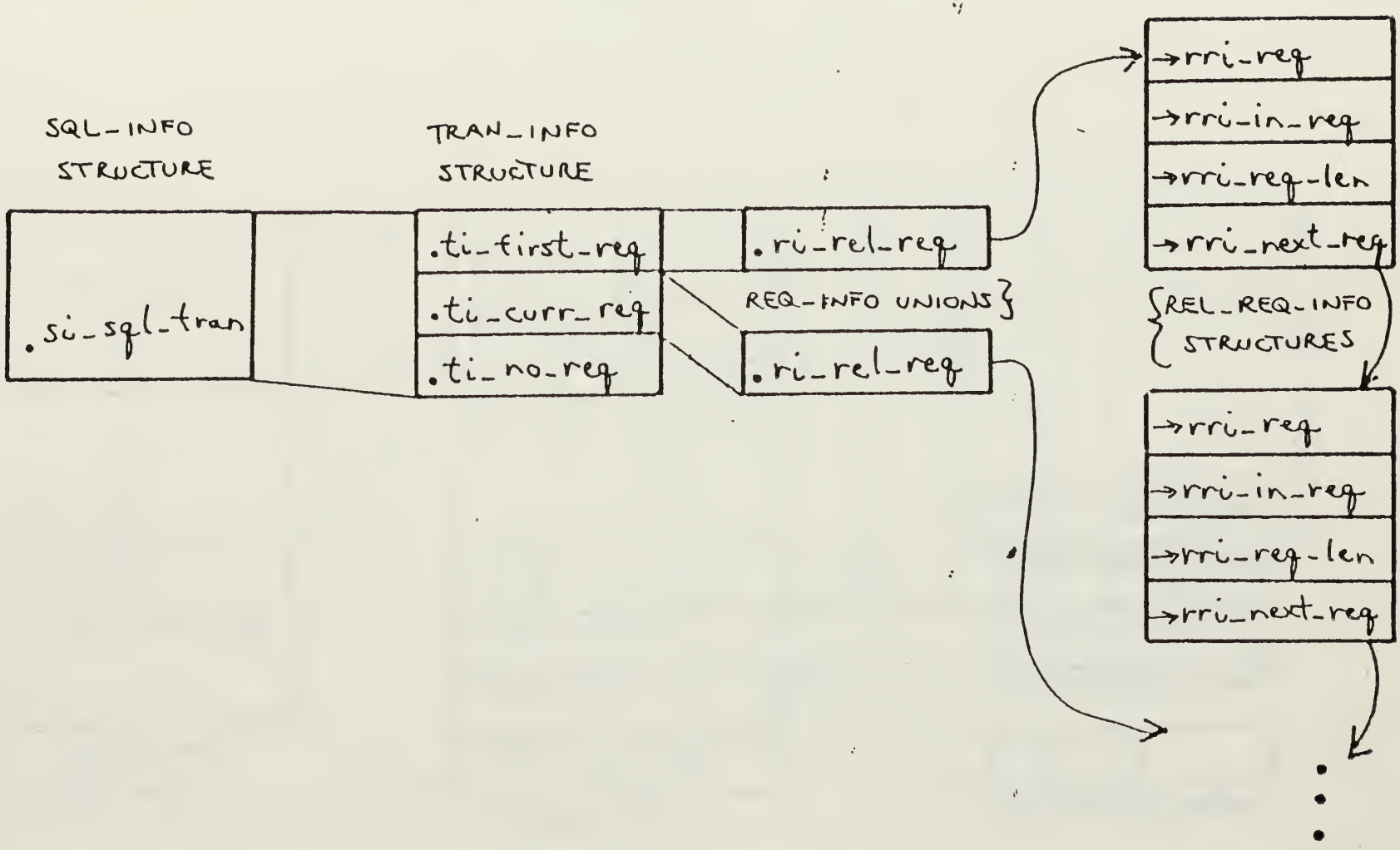


Figure 26. (Continued)

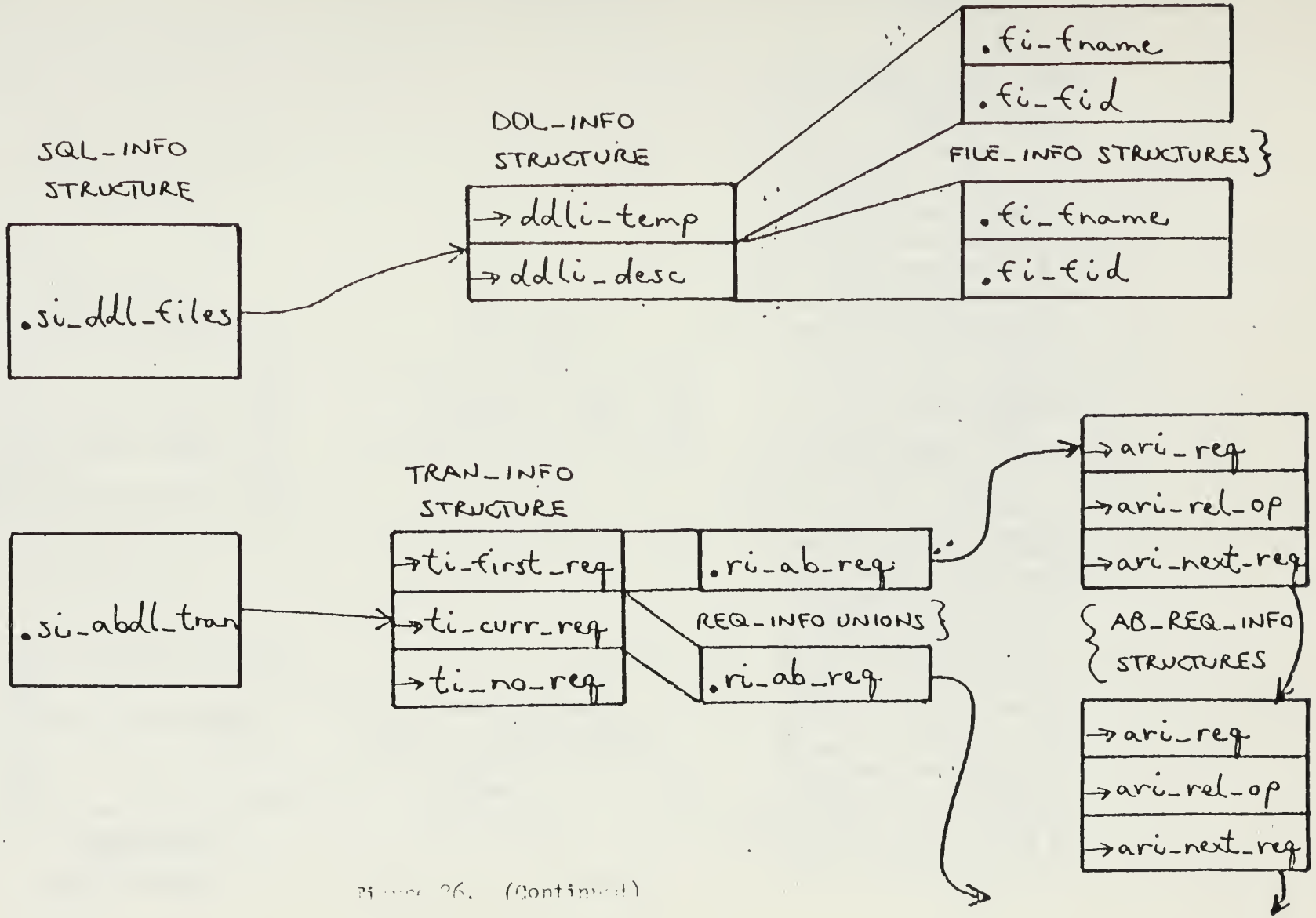
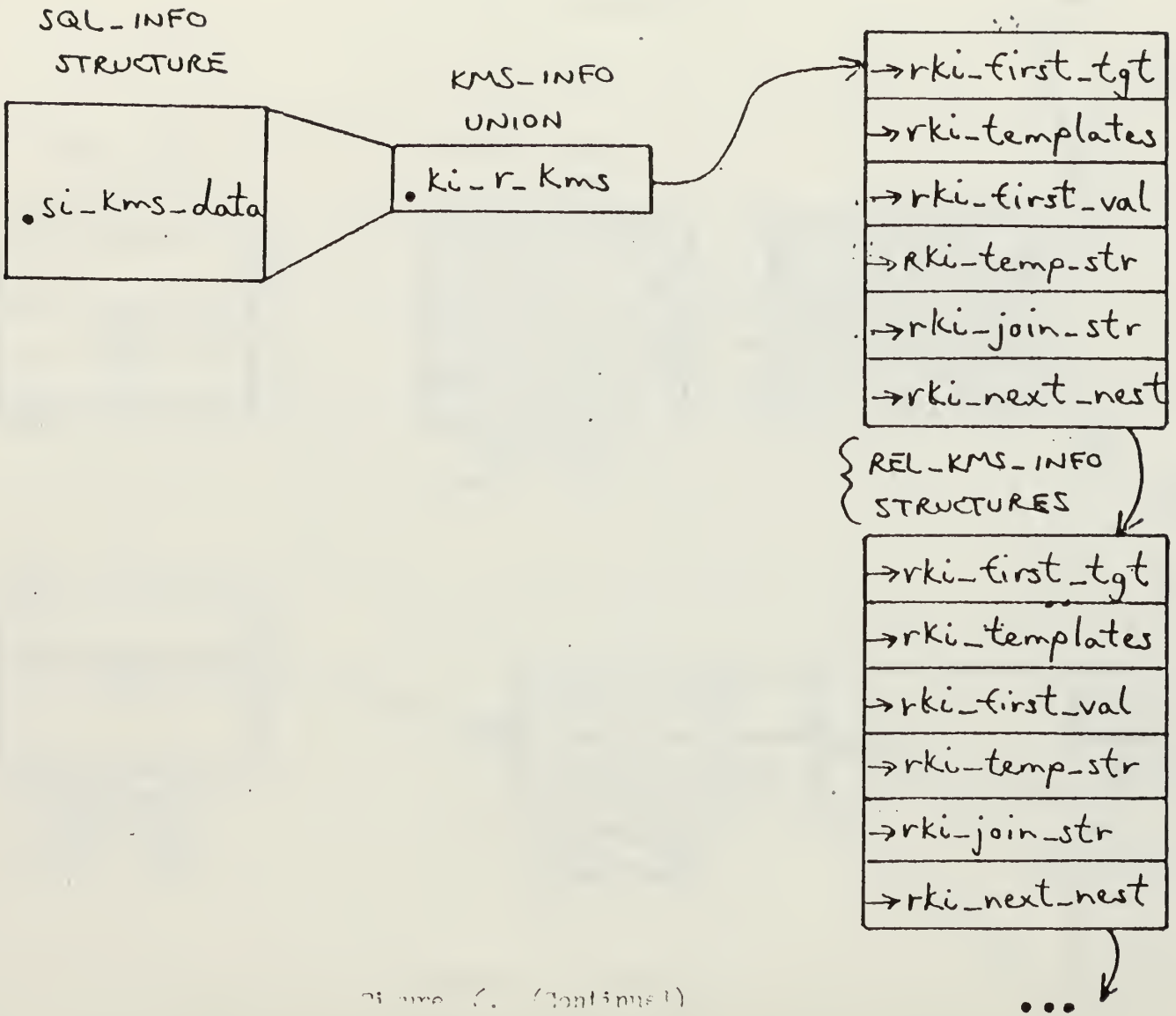


Figure 26. (Continued)


SQL_INFO
STRUCTURE
si_kms_data
KMS_INFO
UNION
ki_r_Kms
→rki_first_tgt
→rki_templates
→rki_first_val
→RKi-temp_str
→rki_join_str
→rki_next_nest
REL_KMS_INFO
STRUCTURES
→rki_first_tgt
→rki_templates
→rki_first_val
→rki_temp_str
→rki_join_str
→rki_next_nest
...

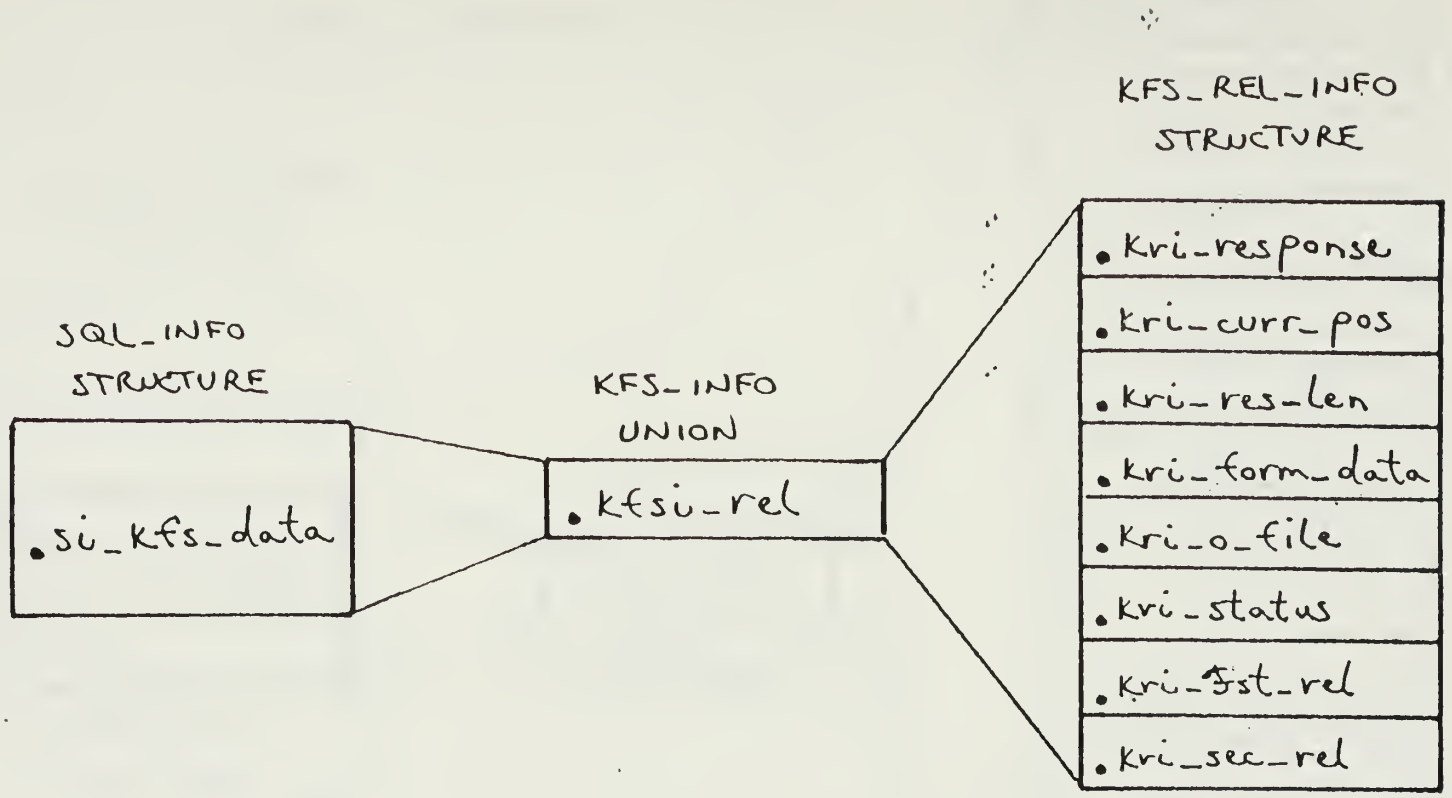


Figure 26. (Continued)

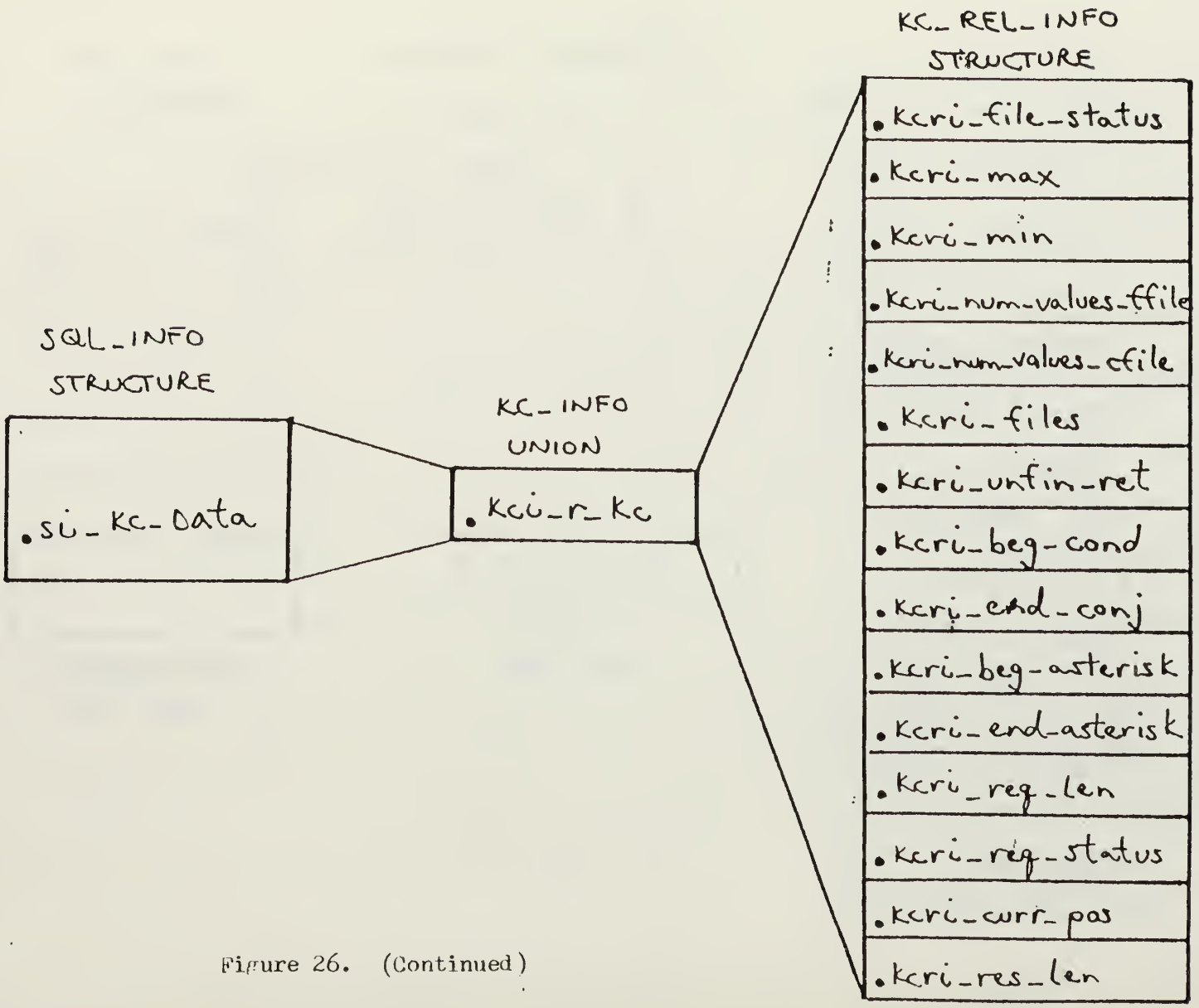


Figure 26. (Continued)

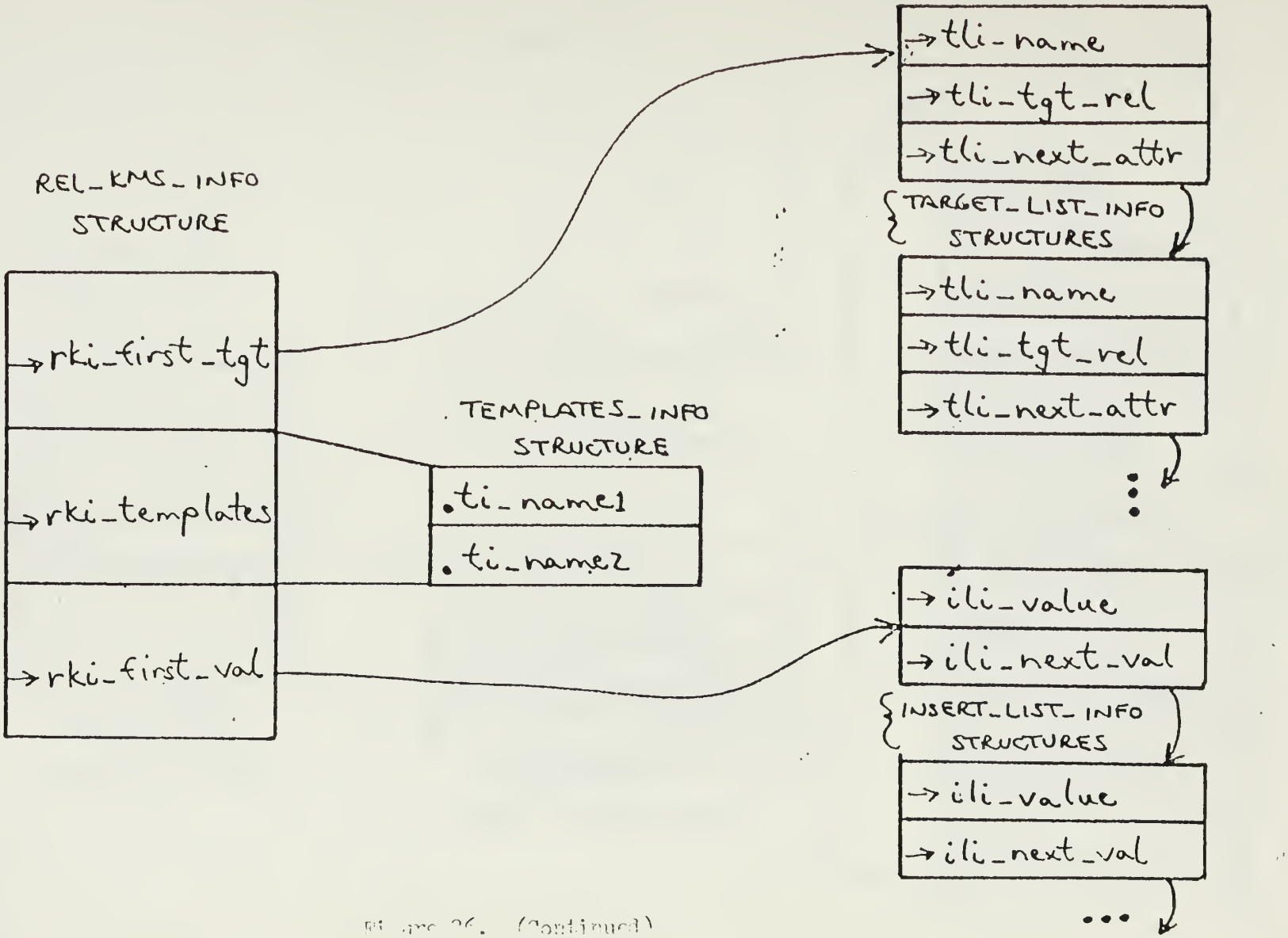


Figure 26. (Continued)

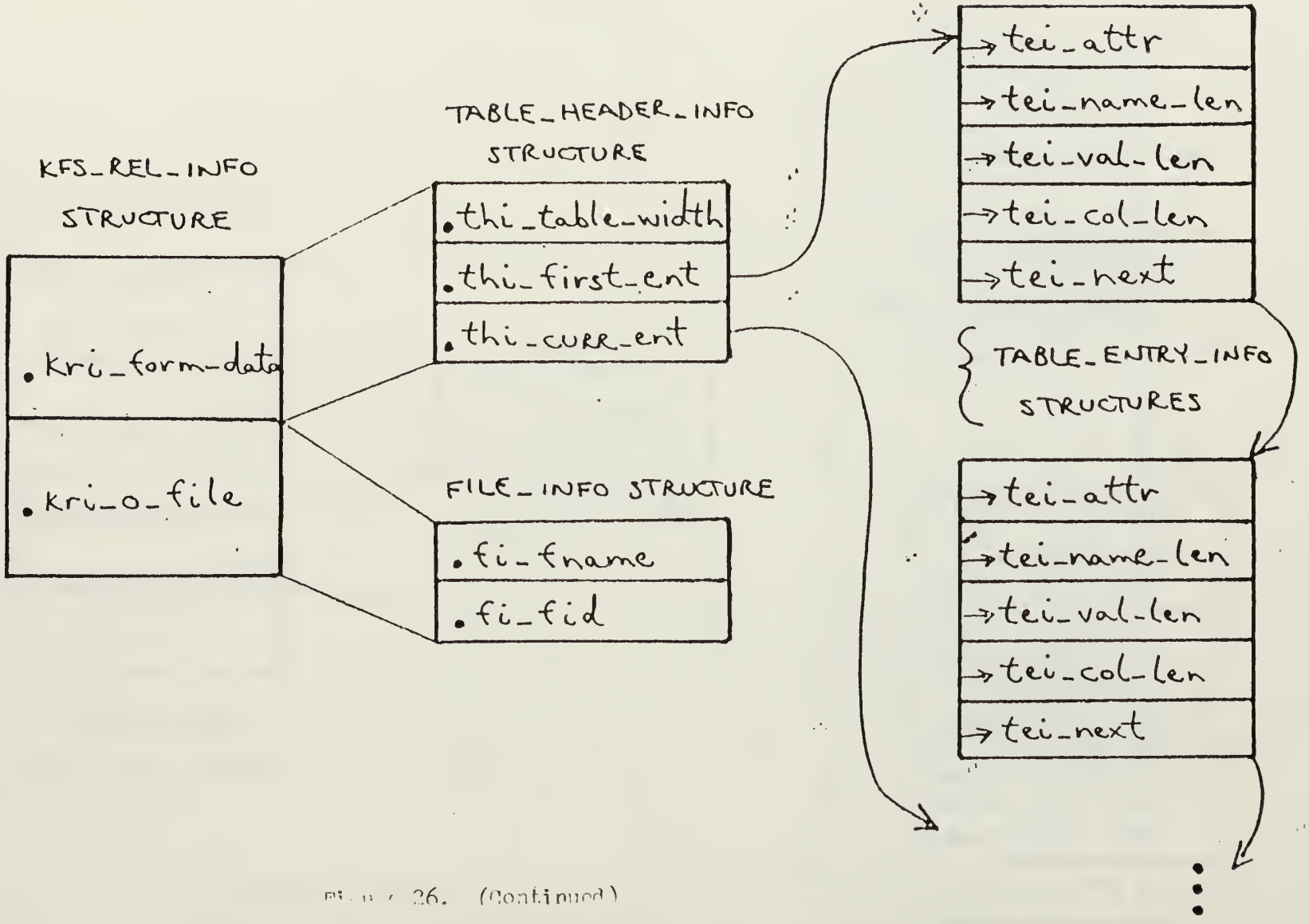


Figure 26. (Continued)

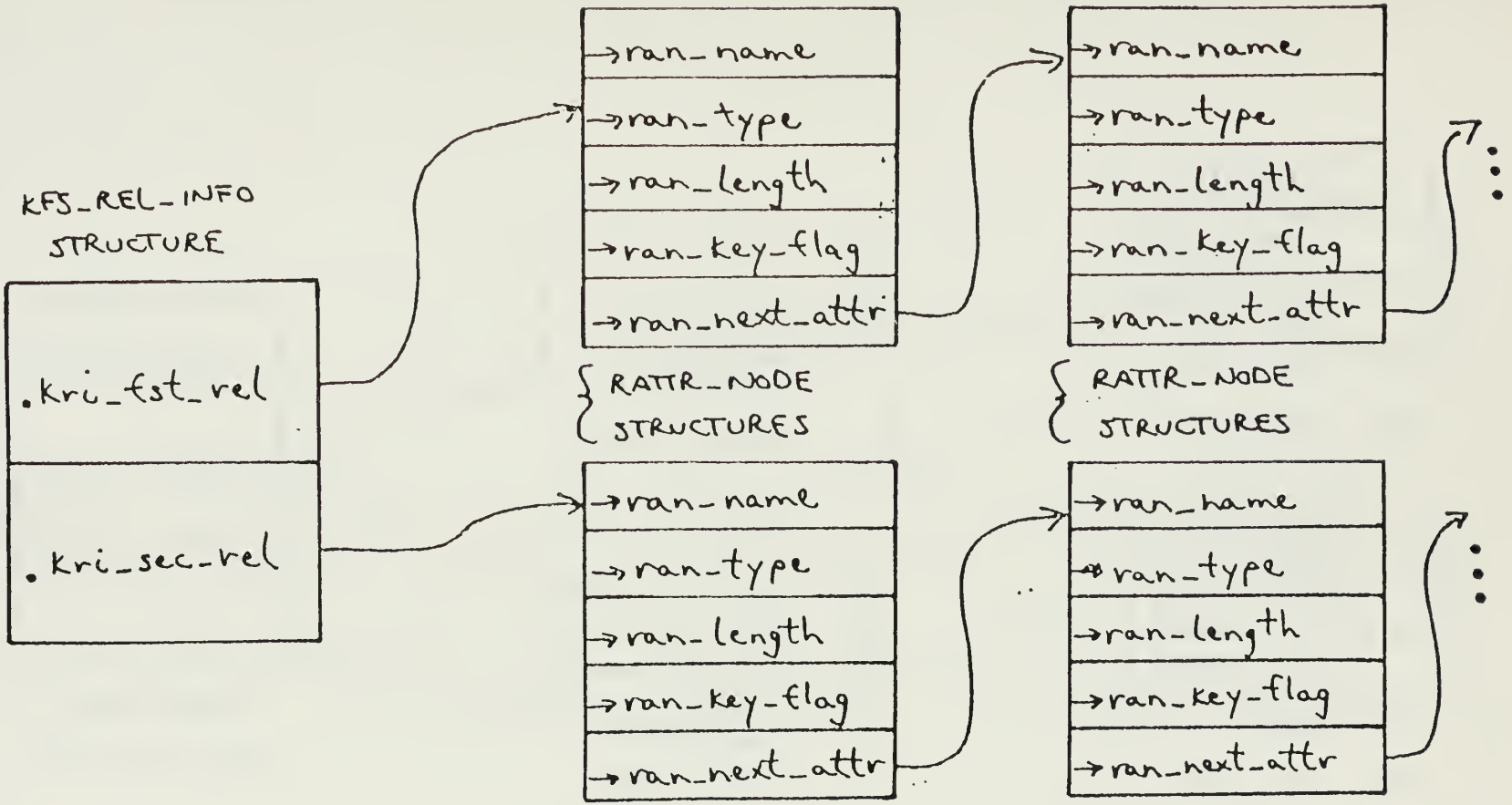
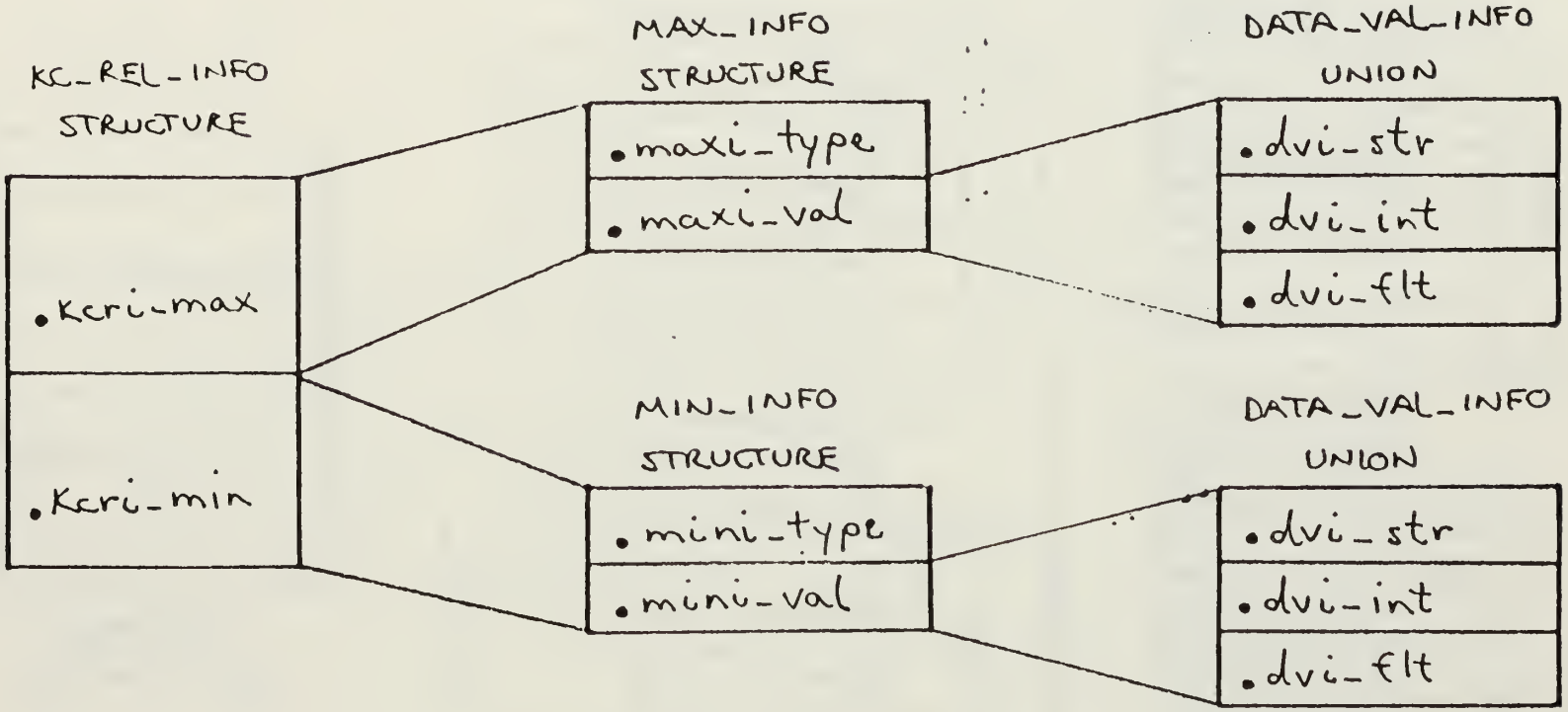
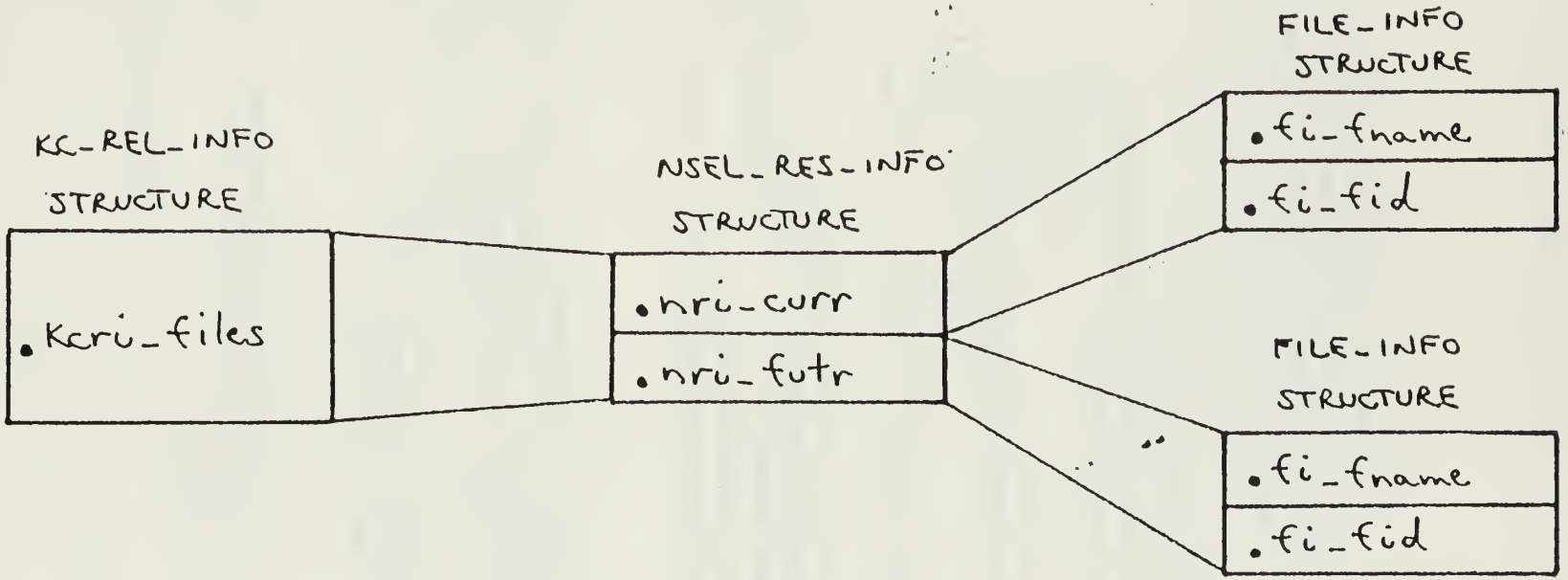


Figure 26. (Continued)


KC_REL_INFO
STRUCTURE
•Kcri_max
•Kcri_min
MAX_INFO
STRUCTURE
•maxi_type
•maxi_val
MIN_INFO
STRUCTURE
•mini_type
•mini_val
DATA_VAL_INFO
UNION
•dvi_str
•dvi_int
•dvi_flt
DATA_VAL_INFO
UNION
•dvi_str
•dvi_int
•dvi_flt


KC-REL-INFO
STRUCTURE
•Kcri_files
NSEL_RES-INFO
STRUCTURE
•nri_curr
•nri_futr
FILE-INFO
STRUCTURE
•fi_fname
•fi_fid
FILE-INFO
STRUCTURE
•fi_fname
•fi_fid

```
module SQL-INTERFACE

    db-list : list;        /* list of existing relational schemas       */
    head-db-list-ptr: ptr; /* ptr to head of the relational schema list */
    current-ptr: ptr;      /* ptr to the current db schema in the list  */
    follow-ptr: ptr;       /* ptr to the previous db schema in the list */
    db-id : string;        /* string that identifies current db in use  */


proc LANGUAGE-INTERFACE-LAYER();
    /* This proc allows the user to interface with the system.  */
    /* Input and output: user SQL requests                      */

      stop : int;  /* boolean flag */
      answer: char;  /* user answers to terminal prompts */

 perform SQL-INIT();
 stop = 'false';
 while (not stop) do
  /* allow user choice of several processing operations */
  print ("Enter type of operation desired");
  print ("    (l) - load new database");
  print ("    (p) - process existing database");
  print ("    (x) - return to the to operating system");
  read (answer);

  case (answer) of
   'l': /* user desires to load a new database */
        perform LOAD-NEW();
   'p': /* user desires to process an existing database */
        perform PROCESS-OLD();
   'x': /* user desires to exit to the operating system */
        /* database list must be saved back to a file   */
        store-free-db-list(head-db-list, db-list);
        stop = 'true';
        exit();
   default: /* user did not select a valid choice from the menu */
          print ("Error - invalid operation selected");
          print ("Please pick again")'
  end-case;

  /* return to main menu */
 end-while;

end-proc;
```

```
proc SQL-INIT();

end-proc;


proc LOAD-NEW();
    /* This proc accomplishes the following:                    */
    /* (1) determines if the new database name already exists.  */
    /* (2) adds a new header node to the list of schemas,       */
    /* (3) determines the user input mode (file/terminal),      */
    /* (4) reads the user input and forwards it to the parser, and */
    /* (5) calls the routine that builds the template/descriptor files */

    answer: int;       /* user answer to terminal prompts */
    more-input: int;   /* boolean flag */
    proceed: int;      /* boolean flag */
    stop : int;        /* boolean flag */
    db-list-ptr: ptr;  /* pointer to the current database */
    req-str: str;      /* single create in SQL form */
    ptr-abdl-list: ptr; /* ptr to a list of ABDL queries (nil for this proc)*/
    tfid, dfid: ptr;   /* pointers to the template and descriptor files */


 /* prompt user for name of new database */
 print ("Enter name of database");
 readstr (db-id);
 db-list-ptr = head-db-list-ptr;

 stop = 'false';
 while (not stop) do
  /* determine if new database name already exists */
  /* by traversing list of relational db schemas   */
  if (db-list-ptr.db-id = existing db) then
    print ("Error - db name already exists");
    print ("Please reenter db name");
    readstr (db-id);
    db-list-ptr = head-db-list-ptr;
  end-if;
  else
    if (db-list-ptr + 1 = 'nil') then
      stop = 'true';
    else
      /* increment to next database */
      db-list-ptr = db-list-ptr + 1;
  end-else;

 end-while;
```

```
/* continue - user input a valid 'new' database name */
/* add new header node to the list of schemas and fill-in db name */
/* append new header node to db-list */
create-new-db(db-id);

/* the KMS takes the SQL creates and builds a new list of relations  */
/* for the new database. After all of the creates have been processed */
/* the template and descriptor files are constructed by traversing  */
/* the new database definition (schema).                            */

more-input = 'true';
while (more-input) do
  /* determine user's mode of input */
  print ("Enter mode of input desired");
  print ("    (f) - read in a group of creates from a file");
  print ("    (t) - read in a single create from the terminal");
  print ("    (x) - return to the main menu");
  read (answer);

  case (answer) of
    'f': /* user input is from a file */
         perform READ-TRANSACTION-FILE();
         perform CREATES-TO-KMS();
         perform FREE-REQUESTS();
         perform BUILD-DDL-FILES();
         perform KERNEL-CONTROLLER();

    't': /* user input is from the terminal */
         perform READ-TERMINAL();
         perform CREATES-TO-KMS();
         perform FREE-REQUESTS();
         perform BUILD-DDL-FILES();
         perform KERNEL-CONTROLLER();

    'x': /* exit back to LIL */
         more-input = 'false';
    default: /* user did not select a valid choice from the menu */
         print ("Error - invalid input mode selected");
         print ("Please pick again");
  end-case;
end-while;

end proc;
```

```
proc  PROCESS-OLD();
     /*  This proc accomplishes the following:                  */
     /*  (1) determines if the database name already exists,    */
     /*  (2) determines the user input mode (file/terminal),    */
     /*  (3) reads the user input and forwards it to the parser */

     answer: int;        /* user answer to terminal prompts */
     found: int;         /* boolean flag to determine if db name is found */
     more-input: int:    /* boolean flag to return user to LIL */
     proceed: int;       /* boolean flag to return user to mode menu */
     db-list-ptr: ptr;   /* pointer to the current database */
     req-str: str;       /* single query in SQL form */
     ptr-abdl-list: ptr; /* pointer to a list of queries in ABDL form */
     tfid, dfid: ptr;    /* pointers to the template and descriptor files */

  /* prompt user for name of existing database */
  print ("Enter name of database");
  readstr (db-id);
  db-list-ptr = head-db-list-ptr;

  found = 'false';
  while (not found) do
   /* determine if database name does exist    */
   /* by traversing list of relational schemas */
   if (db-id = existing db) then
     found = 'true';
   end-if;
   else
     db-list-ptr = db-list-ptr + 1;
     /* error condition causes end of list('nil') to be reached */
     if (db-list-ptr = 'nil') then
       print ("Error - db name does not exist");
       print ("Please reenter valid db name");
       readstr (db-id);
       db-list-ptr = head-db-list-ptr;
     end-if;

   end-else;

  end-while;
```

```
/* continue - user input a valid existing database name */
/* determine user's mode of input */

more-input = 'true';
while (more-input) do
  print ("Enter mode of input desired"):
  print ("    (f) - read in a group of queries from a file");
  print ("    (t) - read in a single query from the terminal");
  print ("    (x) - return to the previous menu");
  read (answer);

  case (answer) of
    'f': /* user input is from a file */
         perform READ-TRANSACTION-FILE();
         perform QUERIES-TO-KMS();
         perform FREE-REQUESTS();

    't': /* user input is from the terminal */
         perform READ-TERMINAL();
         perform QUERIES-TO-KMS();
         perform FREE-REQUESTS();

    'x': /* user wishes to return to LIL menu */
         more-input = 'false';

    default: /* user did not select a valid choice from the menu */
         print ("Error - invalid input mode selected");
         print ("Please pick again");
  end-case;

end-while;

end-proc;
```

```
proc  READ-TRANSACTION-FILE();
  /* This routine opens a create/query file and reads the requests */
  /* into the request list.  If open file fails, loop until valid  */
  /* file entered                                                  */

  while (not open file) do
    print ("Filename does not exist");
    print ("Please reenter a valid filename");
    readstr ( file);
  end-while;

  READ-FILE();

end-proc;

proc READ-FILE();
    /* This routine reads transactions from either a file or the */
    /* terminal into the user's request list structure so that   */
    /* each request may be sent to the KERNEL-MAPPING-SYSTEM.    */

end-proc;


proc  READ-TERMINAL();
    /* This routine substitutes the STDIN filename for the read   */
    /* command so that input may be intercepted from the terminal */

end-proc;


proc CREATES-TO-KMS();
    /* This routine sends the request list of creates one by one */
    /* to the KERNAL-MAPPING-SYSTEM                              */

  while (more-creates) do
    KERNAL-MAPPING-SYSTEM();
  end-while;

end-proc;
```

```
proc QUERIES-TO-KMS();
    /* This routine causes the queries to be listed to the screen. */
    /* The selection menu is then displayed allowing any of the    */
    /* queries to be executed.                                     */

  perform LIST-QUERIES();
  proceed = 'true';
  while (proceed) do
   print ("Pick the number or letter of the action desired");
   print ("    (num) - execute one of the preceding queries");
   print ("    (d)   - redisplay the file of queries");
   print ("    (x)   - return to the previous menu");
   read (answer);

   case (answer) of
    'num' : /* execute one of the queries */
          traverse query list to correct query;
          perform KERNAL-MAPPING-SYSTEM();
          perform KERNEL-CONTROLLER();

    'd'   : /* redisplay queries */
          perform LIST-QUERIES();

    'x'   : /* exit to mode menu */
          proceed = 'false';

    default : /* user did not select a valid choice from the menu */
          print (" Error - invalid option selected");
          print (" Please pick again");
   end-case;

  end-while;

end-proc;

end-module;
```

```
module KMS ()

  perform parser()

end-module KMS



proc  yyparse ()

    /*  This proc accomplishes the following :                    */
    /*  (1) parses the SQL input requests and maps them to appropriate */
    /*      abdl requests, using LEX and YACC to build proc yyparse(). */
    /*  (2) builds the relational schema, when loading a new database. */
    /*  (3) checks for validity of relation and attribute names within */
    /*      the given db schema, when processing requests against an   */
    /*      existing database.                                         */

%{
     list: tgt-list           /* list of attribute names */
     list: templates           /* relation name(s) */
     list: insert-list        /* list of values for insertion op */
     string: temporary-str     /* used for accumulation of query conjuncts */
     string: abdl-str         /* used for accumulation of abdl request */
     string: join-str         /* used for accumulation of join request */
     boolean: nested           /* signals a nested SELECT query */
     boolean: creating         /* signals a DbLoad - versus a DbQuery */
     boolean: or-where         /* signals an OR term in the WHERE clause */
     boolean: and-where        /* signals an AND term in the WHERE clause */
     boolean: set-member       /* signals set membership op, vice nested SEL */
     boolean: common-attr      /* signals COMMON attr predicate of JOIN op */
     boolean: rel1            /* signals curr predicate assoc'd w/1st join rel */
     boolean: rel2            /* signals curr predicate assoc'd w/2nd join rel */
     boolean: or-abdl-join     /* OR in 1st join retrieve request */
     boolean: or-kms-join      /* OR in 2nd join retrieve request */
     boolean: delete-all      /* signals deletion of all records in relation */
     int: target-list-length
     int: insert-list-length
     int: no-templates
     int: no-attributes
     int: attr-len
     char: attr-type
     char: db[]
     char: template[]
     char: attribute[]
%}
```

```
% start  statement

% token    /* LIST ALL TOKENS FROM "LEX", and their TYPE, HERE */

%%

statement:  query
          {
          nested = FALSE
          free all tgt/insert lists and temp-str (malloc'd vars)
          return
          }
        | dml-statement
          {
          cat End-Of-Request ("]") to end of abdl-str
          free all tgt/insert lists and temp-str (malloc'd vars)
          return
          }
        | ddl-statement
          {
          return
          }
          ;

dml-statement:  insertion
             |  deletion
             |  update
                ;

query:  query-expr
      ;

query-expr:  query-block
           {
           cat End-Of-Request ("]") to end of abdl-str
           }
           ;
```

```
query-block: select-clause  FROM  from-list
             {
             for (ea attribute name in tgt-list)
               if  (! join)
                 if  NOT valid-attribute(db, template, attribute, attr-len)
                 print ("Error - field name 'attribute-name' does not exist")
                   perform yyerror()
                   return
                 end-if
               end-if
               else
                 a join exists --  check that tgt-rel(s) match at least
                 one from-list relation
                 if  (match neither)
                   print ("Error - 'attr' attr not in from-list relations")
                   perform yyerror()
                   return
                 end-if
               end-else
             end-for
             cat "(" to abdl-str
             if  (join)
               cat "(" to join-str
             end-if
             if  (nested)
               fill temporary-str w/'*'s marking the length of the tgt attr
             end-if
             }
           A
             {
             cat ")" to abdl-str
             if  (! join)
               cat "('tgt-list')" to abdl-str
             end-if
             else
               cat "('tgt-list')" to abdl/join-str, as appropriate
               construct the rest of the abdl join request
               (ie, cat COMMON-str to abdl-str; cat join-str to abdl-str)
             end-else
             }
           B
             ;
```

```
A:  empty
      {
      cat "FILE = 'relation-name'" to abdl-str
      }
  |   WHERE boolean
      {
      if  (! join) && (or-where)
        cat ")" to abdl-str
      end-if
      else if  (or-abdl-join)
        cat ")" to abdl-str
      end-elseif
      elseif  (or-kms-join)
        cat ")" to join-str
      end-elseif
      }
      ;

B:  empty
  |   GROUP BY  field-spec-list
      {
      cat "BY 'attribute-name'" to abdl-str
      }
      ;

select-clause:  SELECT
                {
                if  (nested)
                  allocate another set of tgt/insert lists, temporary-str,
                  and abdl strings
                end-if
                copy "[ RETRIEVE  " to beginning of abdl-str
                }
              C
                ;

C:  sel-expr-list
  |   MULTOP
      {
      /* retrieval of "all" attribute values desired */
      if (MULTOP value /= '*')
        print ("Error - asterisk(*) operator expected")
        perform yyerror()
        return
      end-if
      }
      ;
```

```
sel-expr-list:  sel-expr
                {
                copy first attribute name to tgt-list
                }
              | sel-expr-list  COMMA  sel-expr
                {
                copy successive attribute name(s) to tgt-list
                }
                ;

sel-expr:  expr
           ;

insertion:  INSERT INTO
            {
            copy "[ INSERT (" to beginning of abdl-str
            }
           receiver  COLON  insert-spec
            {
            cat ")" to abdl-str
            }
            ;

receiver:  table-name
           {
           cat "<FILE, 'relation-name'>" to abdl-str
           }
         D
           ;
```

```
D: empty
    {
    /* inserting info for "all" attribute values */
    copy all attribute names from schema to tgt-list
    if (target-list-length < 1)
      print ("Error - rel does not exist, or has no attr's")
      perform yyerror()
      return
    end-if
    }
  | LPAR field-name-list RPAR
    {
    for (ea attribute name in tgt-list)
      if NOT valid-attribute(db, template, attribute, attr-len)
        print ("Error - field name 'attribute-name' does not exist)
        perform yyerror()
        return
      end-if
    end-for
    }
    ;

field-name-list: field-name
                 {
                 target-list-length++
                 copy first attribute name to tgt-list
                 }
               | field-name-list COMMA field-name
                 {
                 target-list-length++
                 copy successive attribute name(s) to tgt-list
                 }
                 ;

insert-spec: literal
             {
             if (length of tgt-list <> length of insert-list)
               print ("Error - not enough or too many values inserted")
               perform yyerror()
               return
             end-if
             for (ea attribute in tgt-list / ea value in insert-list)
               perform type-checking of attrribute-value pairs
               cat ",<'attribute-name', 'insert-value'>" to abdl-str
             end-for
             }
             ;
```

```
deletion: DELETE table-name
          {
          copy "[ DELETE (  " to abdl-str
          copy 'table-name' to templates
          }
        E
          {
          if (delete-all)
            cat "TEMPLATE = 'table-name'" to abdl-str
          end-if
          cat ")" to abdl-str
          }
          ;

E: empty
    {
    delete-all = TRUE
    }
 | WHERE boolean
    {
    if (or-where)
      cat ")" to abdl-str
    end-if
    }
    ;

update: UPDATE table-name
        {
        copy "[ UPDATE (  " to beginning of abdl-str
        copy relation-name to templates
        }
       set-clause-list  F
        {
        cat ") 'set-clause-list'" to abdl-str
        }
        ;

F: empty
 | WHERE boolean
    {
    if (or-where)
      cat ")" to abdl-str
    end-if
    }
    ;
```

```
set-clause-list:  set-clause
                  ;

set-clause:  SET  field-name  EQ  expr
             {
             if  NOT validattribute(db, template, attribute, attr-len)
               print ("Error - field name 'attribute-name' does not exist")
               perform yyerror()
               return
             end-if
             else
               copy "<'field-name = expr'>" to abdl-str
             end-else
             }
             ;

ddl-statement:  create-table
                ;

create-table:  CREATE
               {
               creating = TRUE
               locate db-id schema header
               }
               TABLE  table-name COLON
               {
               no-templates ++
               create new template block
               enter 'relation-name' in template block
               }
               field-defn-list
               ;

field-defn-list:  field-defn
                  {
                  no-attributes ++
                  }
                | field-defn-list  COMMA  field-defn
                  {
                  no-attributes ++
                  }
                  ;
```

```
field-defn: field-name LPAR type G RPAR
          {
          create new attribute block
          enter 'attribute-name' in attribute block
          }
          ;

G: empty
    {
    set key-flag to '0' in attribute block
    }
  | COMMA NONULL
    {
    set key-flag to '1' in attribute block
    }
    ;

type: CHAR LPAR INTEGER RPAR
      {
      enter attribute type and length in attribute block
      }
    | INT LPAR INTEGER RPAR
      {
      enter attribute type and length in attribute block
      }
    | FLOAT LPAR INTEGER RPAR
      {
      enter attribute type and length in attribute block
      }
      ;
```

```
boolean:  boolean-term
          {
          if  (! join)
            cat "-(FILE = 'relation-name') and" to abdl-str
            cat temporary-str to abdl-str
          end-if
          }
        | boolean  OR
          {
          or-where = TRUE
          if  (! join)
            abdl-str[11] = '('
            cat ") or ((FILE = 'relation-name') and" to abdl-str
            copy empty-str to temporary-str
          end-if
          }
          boolean-term
          {
          if  (! join)
            cat temporary-str to abdl-str
          end-if
          else
            if  (current predicate assoc'd w/same rel as previous predicate)
              abdl/join-str[11] = '('
              cat ") or ((FILE = 'rel-name') and" to abdl/join-str (as approp)
              cat temporary-str to appropriate str (abdl/join-str)
            end-if
            else
              abdl/join-str(as approp)[11 + 3] = '('
              cat "and" to appropriate str (abdl/join-str)
              cat temporary-str to appropriate str (abdl/join-str)
            end-else
            copy empty str to temporary-str
            or-where = FALSE
          end-else
          }
          ;
```

```
boolean-term:  boolean-factor
               {
               if  (join) && (! or-where)
                 determine rel that curr predicate is assoc'd with
                 if  (rel1) && (! common-attr)
                   cat "(FILE = 'rel-name1') and" to abdl-str
                   cat temporary-str to abdl-str
                   cat " FILE = 'rel-name2'" to join-str
                 end-if
                 if  (rel2) && (! common-attr)
                   cat "(FILE = 'rel-name2') and" to join-str
                   cat temporary-str to join-str
                   cat " FILE = 'rel-name1'" to abdl-str
                 end-if
                 if  (common-attr)
                   cat " FILE = 'rel-name1/2" to abdl/join-str's
                 end-if
               end-if
               }
             | boolean-term  AND
               {
               and-where = TRUE;
               if  (! join)
                 cat "and" to temporary-str
               end-if
               }
               boolean-factor
               {
               if  (join) && (! or-where) && (! common-attr)
                 if  (rel1)
                   abdl-str[11 + 3] = '('
                   cat ") and" to abdl-str
                   cat temporary-str to abdl-str
                 end-if
                 if  (rel2)
                   join-str[11 + 3] = '('
                   cat ") and" to join-str
                   cat temporary-str to join-str
                 end-if
                 copy empty-str to temporary-str
                 and-where = FALSE
               end-if
               }
               ;
boolean-factor:  boolean-primary
                 ;
```

```
boolean-primary:  predicate
                  ;

predicate:  expr
            {
            if  (! join)
              if  NOT valid-attribute(db, template, attribute, attr-len)
                print ("Error - field name 'attribute-name' does not exist")
                perform yyerror()
                return
              end-if
              if  (! and-where)
                allocate new temporary-str
              end-if
              cat "('attribute-name' " to temporary-str
              and-where = FALSE
            end-if
            else
              save 'type' for later comparison during type-checking,
              in case this is the COMMON attribute predicate
            end-else
            }
          comparison
            {
            if  (nested)
              save attr name in case nest is actually a set membership op
            end-if
            }
          table-spec
            {
            if  (! join)
              cat ")" to temporary-str
            end-if
            else
              if  (common-attr)
                save values of 'expr', 'comparison', & 'table-spec'
                for the COMMON expr,  and type-check the two attr's
              end-if
              if  (! and-where) && (! or-where)
                allocate initial temporary-str
                copy "(" to temporary-str
              end-if
              else
                cat "(" to temporary-str
              end-else
              cat "'expr' 'comparison' 'table-spec')" to temporary-str
            end-else
            }
            ;
```

```
comparison: comp-op
          {
          if (! join)
            cat 'comp-op' to temporary-str
            if (nested)
              copy type-op-code to abdl-str.rel-op
            end-if
          end-if
          }
          ;

comp-op: EQ
     | M J
         {
         if (nested)
           cat 'J' to 'M' and save
         end-if
         }
     | L
         {
         nested = TRUE
         }
         ;

J: empty
 | K
    {
    nested = TRUE
    }
    ;

K: ANY | ALL
    ;

L: IN | NOT IN
    ;

M: NE | RWEDGE | GE | LWEDGE | LE
    ;
```

```
table-spec: literal
            {
            if (! set-member)
              if ('literal[0]' = QUOTE)
                strip quotes from literal
                change literal to ALPHANUMFIRST
                literal-const = FALSE
              end-if
              cat result, or original literal, to temporary-str
              if (nested)
                set first-ptr to top of abdl-str list
              end-if
            end-if
            else
              set-member = FALSE
            end-else
            }
          | query-expr
            {
            increment ptr to next tgt/insert list, temp-str, and abdl-str
            }
          | LPAR query-expr RPAR
            {
            increment ptr to next tgt/insert list, temp-str, and abdl-str
            }
          | expr
            {
            common-attr = TRUE
            }
            ;
```

```
literal: lit-tuple
    | LPAR entry-list RPAR
      {
      set-member = TRUE
      case (set-membership-op)
       3,5.8.10 :          /* <=ANY, <ANY, >=ALL, >ALL */
            cat 'max of value set' to temporary-str
       4,6,7,9 :           /* >=ANY, >ANY, <=ALL, <ALL */
            cat 'min of value set' to temporary-str
       1 :                 /* NOT IN */
          cat first value to temporary-str
          while (other values exist)
           cat ") and ('attr-name' /= 'value'" to temporary-str
          end-while
       0,2 :               /* IN, /=ANY */
          cat first value to temporary-str
          if (more values exist)
           abdl-str[11] = '('
           or-where = TRUE
          end-if
          while (other values exist)
           cat ")) or ((TEMPLATE = 'rel-name') and ('attr-name'"
           to temporary-str
           if ( rel-op = IN )
            cat " = " to temporary-str
           end-if
           else
            cat " /= " to temporary-str
           end-else
           cat value to temporary-str
          end-while
      end-case
          ;

lit-tuple: entry
    | LWEDGE entry-list RWEDGE
      ;
```

```
entry-list: entry
            {
            /* copy first value to insert-list */
            insert-list-length++
            if ('entry[0]' = QUOTE)
              strip quotes from entry
              change entry to ALPHANUMFIRST
            end-if
            copy result, or original entry, to insert-list
            }
          | entry-list  COMMA  entry
            {
            /* copy successive value(s) to insert-list */
            insert-list-length++
            if ('entry[0]' = QUOTE)
              strip quotes from entry
              change entry to ALPHANUMFIRST
            end-if
            copy result, or original entry, to insert-list
            }
            ;

entry: constant
       ;

expr: arith-term
    | expr  ADDOP  arith-term
      ;

arith-term: arith-factor
          | arith-term  MULT-OP  arith-factor
            ;

arith-factor: H  primary
              ;

H: empty
 | ADDOP
   ;

primary: field-spec
       | set-fn  LPAR  field-name  RPAR
       | LPAR  expr  RPAR
       | constant
         ;

field-spec-list: field-spec
                 ;
```

```
field-spec:  field-name
        | table-name  DOT  field-name
            {
            if  (! valid-attribute(db, rel, attr, attr-len)
              print ("Error - 'rel.attr' is invalid combination")
              perform yyerror()
              return
            end-if
            if  (join)
              if  (! or-where) || ( (or-where) && (! and-where) )
                if  (table-name = rel1)
                  rel1 = TRUE
                  rel2 = FALSE
                end-if
                if  (table-name = rel2)
                  rel1 = FALSE
                  rel2 = TRUE
                end-if
              end-if
            end-if
            }
            ;

set-fn:  AVG  |  MAX  |  MIN  |  SUM  |  COUNT
        ;

from-list:  table-name
            {
            copy first relation name to templates
            if  (tgt-list = null)
              fill tgt-list with "all" attribute names in the relation
            end-if
            }
        |  from-list  COMMA  table-name
            {
            copy second relation name to templates
            join = TRUE
            allocate join-str
            }
            :

empty:  :
```

```
constant: QUOTE I QUOTE
          {
          literal-const = TRUE
          perform type-checking
          }
        | INTEGER
          {
          perform type-checking
          }
          ;

I: IDENTIFIER
 | VALUE
    ;

field-name: IDENTIFIER
          ;

table-name: IDENTIFIER
          {
          if (! creating)
            if NOT valid-table(db, template)
              print ("Error - relation name 'table-name' does not exist")
              perform yyerror()
              return
            end-if
          end-if
          }
          ;
%%

end-proc yyparse



proc parser ()
  {
  if (! creating)
    allocate and initialize first tgt/insert lists, temporary-str, and abdl-str
    /* if an old abdl-str exists, free it first */
  end-if
  perform yyparse()
  reset all boolean and counter variables
  }
end-proc parser
```

```
proc  yyerror (s)
  char  *s
  {
  if  (creating)
    set CreateDB-error-flag
    print ("Error msg - tell user which CREATE TABLE request was in error")
    free current schema (malloc'd vars)
  end-if
  else
    free all tgt/insert lists, temp-str, and abdl-strs
  end-else
  reset all boolean and counter variables
  }
end-proc yyerror
```

```
module Kernel-Controller()

 /*  This procedure accomplishes the following:                    */
 /*  (1) Initialization pointers global to the Kernel Controller.  */
 /*  (2) Checks si-operation to determine whether we are creating a */
 /*      database, retrieving information from the database, deleting */
 /*      information from the database, inserting information into the */
 /*      database, updating the database or if there are errors.   */
 /*  (3) Depending on the appropriate operation the corresponding   */
 /*      procedure is called.                                       */

 begin module

  sql-ptr = &(cuser-rel-ptr->ui-li-type.li-sql);
  kc-ptr  = &(sql-ptr->si-kc-data.kci-r-kc);
  /* Initializes pointers global to the kernel controller */

  /* look at the si-operation to determine what action to take */
  case si-operation

    'Create a database':

      perform load-tables();
      break;

    'Execute retrieve requests':

      perform select-requests-handler();
      break;

    'Execute retrieve common requests':
    'Execute delete requests':
    'Execute insert requests':
    'Execute update requests':

      perform rest-requests-handler();
      break;

    'Otherwise':  /* There are errors */

      print error message;
      break;

  end case

 end module
```

```
proc load-tables()

  /*  This procedure accomplishes the following:          */
  /*  (1) Calls dbl-template() which is a procedure       */
  /*      already defined in the Test Interface.  It loads */
  /*      the template file.                              */
  /*  (2) Calls dbl-dir-tbls() which is another procedure  */
  /*      already defined in the Test Interface.  It loads */
  /*      the descriptor file.                            */

  begin proc

    do initialization; /* Initialize pointer */

    perform dbl-template(&template, ptr->ddli-temp.fi-fid);

    perform dbl-dir-tbls(ptr->ddli-desc.fi-fid);

  end proc


proc rest-requests-handler()

  /*  This procedure handles common retrieve requests, insert  */
  /*  requests, delete requests and update requests by calling */
  /*  sql-execute().                                          */

  begin proc

    perform sql-execute();

  end proc
```

```
proc select-requests-handler()

 /*  This procedure accomplishes the following:                    */
 /*  (1) Determines if we have a series of requests which          */
 /*      corresponds to a nested select in SQL.                    */
 /*  (2) If we do have a series of requests we process the first   */
 /*      request because this is the only fully formed ABDL        */
 /*      request.  This is accomplished by calling sql-execute().  */
 /*  (3) If it is a nested select, we enter a loop to process the  */
 /*      remaining requests.  Note that it may be necessary to     */
 /*      process sub-requests.  This requires entering another     */
 /*      loop to process these. This occurs when the number of     */
 /*      responses to a request is larger than NUM-CONJ.  In this  */
 /*      situation a request contains at most NUM-CONJ values.     */
 /*  (4) If it is not a nested select, then only one request       */
 /*      requires processing.  This is accomplished by calling     */
 /*      sql-execute.                                              */

 begin proc
  curr-req = &(sql-ptr->si-abdl-tran->ti-curr-req);
    /* Set curr-req equal to the first request to be processed */
  num-reqs = &(sql-ptr->si-abdl-tran->ti-no-req);
    /* Set num-reqs equal to the number of requests to be processed */
  kc-ptr->kcri-file-status = FIRSTTIME;
    /* Set the file status to indicate it is the first time through */
  kc-ptr->kcri-req-status = FIRSTTIME;
    /* Set the request status to indicate it is the first time through */
  kc-ptr->kcri-num-values-ffile = 0;
    /* Set the number of values in the file to zero */
  strcpy(kc-ptr->kcri-files.nri-futr.fi-fname, CURRFName);
    /* Assigns filename for the current file */
  strcpy(kc-ptr->kcri-files.nri-curr.fi-fname, FUTRFName);
    /* Assigns filename for the future file */

  *num-reqs = *num-reqs - 1; /* Decrement num-reqs */
  if (*num-reqs == 0) /* Its the last subrequest */
    sql-ptr->si-subreq-stat = LASTSUBREQ;
  else  /* Its an intermediate subrequests */
    sql-ptr->si-subreq-stat = INTERSUBREQ;

  perform sql-execute(); /* Handles the first request */
```

```
 while (*num-reqs > 0)  /* It is a nested select */
  begin while
   *num-reqs = *num-reqs - 1; /* Decrement num-reqs */
   perform swap-files(); /* Swap current and future files */
   kc-ptr->kcri-num-values-cfile = kc-ptr->kcri-num-values-ffile;
    /* Set the number of values in the current file */
   kc-ptr->kcri-num-values-ffile = 0;
    /* Reinitializes the number of values in the future file */
   kc-ptr->kcri-file-status = FIRSTTIME; /* Reintialize the status */
   sql-ptr->si-subreq-stat = INTERSUBREQ; /* Reinitialization the status*/
   curr-req->ri-ab-req = curr-req->ri-ab-req->ari-next-req;
    /* Advance pointer so it points to the next request */
   kc-ptr->kcri-unfin-ret = curr-req->ri-ab-req->ari-req;
    /* Loads abdl request template into unfin-ret */
     curr-req->ri-ab-req->ari-req = NULL;
    /* Sets ari-req to empty so that the completed request can */
    /* be built into ari-req */
   kc-ptr->kcri-req-status = FIRSTTIME; /* Reinitialize request status */
   one-conj-flag = FALSE;
    /* Sets flag to indicate it is not a one conjunction type req */

   while ((kc-ptr->kcri-num-values-cfile > 0) && ( !one-conj-flag ))
    /* There are values left to insert into the request */
    begin while
     perform build-request( &one-conj-flag );
      /* Builds the next request */
     perform sql-execute();
      /* Handles the request just built */
     perform free( curr-req->ri-ab-req->ari-req );
      /* Frees ari-req */
     curr-req->ri-ab-req->ari-req = NULL;
      /* Reinitializes ari-req */
    end while

    /* Sets up for the next request */
   curr-req->ri-ab-req->ari-req = kc-ptr->kcri-unfin-ret;
    /* Set ari-req equal to unfin-ret; */
   kc-ptr->kcri-unfin-ret = NULL;
    /* Reinitailize unfin-ret */
   perform fclose(kc-ptr->kcri-files.nri-curr.fi-fid);
    /* Close the current file */
  end while
end proc
```

```
proc sql-execute()

  /*  This procedure accomplishes the following:                  */
  /*  (1) Sends the request to MBDS using TI-S$TrafUnit()         */
  /*      which is defined in the Test Interface.                 */
  /*  (2) Calls sql-requests-left() to ensure that all requests   */
  /*      have been processed.                                    */
  /*  (3) Calls TI-finish() for post operation processing.        */

  begin proc

    perform TI-S$TrafUnit(sql-ptr->si-curr-db.cdi-dbname,
              sql-ptr->si-abdl-tran->ti-curr-req.ri-ab-req->ari-req);
      /* Sends the request to MBDS */

    perform sql-chk-responses-left();
      /* Wait until all responses have been returned */

    perform TI-finish();
      /* Routine to tidy things up after processing is completed */

  end proc
```

```
proc sql-chk-responses-left()

 /*  This procedure accomplishes the following:                */
 /*  (1) Receives the message from MBDS by calling             */
 /*      TI-R$Message() which is defined in the Test Int.      */
 /*  (2) Gets the message type by calling TI-R$Type.           */
 /*  (3) If not all responses to the request have been         */
 /*      returned, a loop is entered.  Within this loop a      */
 /*      case statement separates the responses received by    */
 /*      message type.                                         */
 /*  (4) If the response contained no errors, then procedure   */
 /*      TI-R$Req-res() is called to receive the response from */
 /*      MBDS.                                                 */
 /*  (5) A check is then made to determine if this is the last */
 /*      response.  If it is, then the results are returned to */
 /*      the calling routine.  If it was not the last response */
 /*      then the results are filed in future-results-file.    */
 /*  (6) If the message contained an error then procedure      */
 /*      TI-R$ErrorMessage is called to get the error message  */
 /*      and then procedure TI-ErrRes-output is called to      */
 /*      output the error mesage.                              */

 begin proc

  num-reqs = &(sql-ptr->si-abdl-tran->ti-no-req);
    /* Number of requests left, not counting the request */
    /* currently being worked on. */

  response = sql-ptr->si-kfs-data.kfsi-rel.kri-response;
    /* Initailize response */

  done = FALSE;  /* Initialize flag */

  while ( not done )
    /* Not all responses for the current request have been received */

    perform TI-R$Message(); /* Receive message from MBDS */

    msg-type = TI-R$Type(); /* Get the type of the received message */
```

```
  case msg-type /* Is the response correct or are there errors? */

   'CH-ReqRes': /* The response is correct */
    done = chk-if-last-response();
     /* Set flag if its the last response */

    case sql-ptr->si-operation

     'Execute Retrieve Requests':
     'Execute Retrieve Common Requests':
      if (*num-reqs == 0)
        /* If there are no requests left, send the results to */
        /* the formatter. */
        Kfs();
      else
        /*There are requests left the in nested select to process */
        file-future-results(); /* Save the results */
       break;

     'Execute Delete Requests':
       print "Delete Query Done";
       break;

     'Execute Insert Requests':
       print "Insert Query Done";
       break;

     'Execute Update Requests':
       print "Modify Query Done";
       break;

    end case
    break;

   'Requests With Errors':
    perform TI-R$ErrorMessage(request,err-msg);
     /* Get the error message */
    perform TI-ErrRes-output(request,err-msg);
     /* Output the error message */
    done = TRUE; /* Set the flag */
    break;

  end case
 end while
end proc
```

```
proc build-request( one-conj-flag )

 /*  This procedure accomplishes the following:                */
 /*  (1) Builds the next ABDL request to be processed by       */
 /*      calling either N-Conjunction, Not-In-Conjunction or   */
 /*      One-conjunction depending on the relational operator. */
 /*  (2) Sets one-conj-flag if procedure One-Conjunction is    */
 /*      called.                                               */

 begin proc

  curr-abdl-req = &(sql-ptr->si-abdl-tran->ti-curr-req);
   /* Gets the current ABDL request */

  case curr-abdl-req->ri-ab-req->ari-rel-op
   /* Switches based upon the relational operator in the request */

    'In Operator':
        perform N-Conjunction();
        break;

    'Not In Operator':
        perform Not-In-Conjunction();
        break;

    'Not Equal to Any Operator':
        perform N-Conjunction();
        break;

    'Less than or Equal to Any Operator':
        perform One-Conjunction(kc-ptr->kcri-max.maxi-val.dvi-int);
        *one-conj-flag = TRUE;
        break;

    'Greater than of Equal to Any Operator':
        perform One-Conjunction(kc-ptr->kcri-min.mini-val.dvi-int);
        *one-conj-flag = TRUE;
        break;

    'Less than Any Operator':
        perform One-Conjunction(kc-ptr->kcri-max.maxi-val.dvi-int);
        *one-conj-flag = TRUE;
        break;
```

```
    'Greater than Any Operator':
        perform One-Conjunction(kc-ptr->kcri-min.mini-val.dvi-int);
        *one-conj-flag = TRUE;
        break;

    'Less than or Equal to All Operator':
        perform One-Conjunction(kc-ptr->kcri-min.mini-val.dvi-int);
        *one-conj-flag = TRUE;
        break;

    'Greater than or Equal to All Operator':
        perform One-Conjunction(kc-ptr->kcri-max.maxi-val.dvi-int);
        *one-conj-flag = TRUE;
        break;

    'Less than All Operator':
        perfrom One-Conjunction(kc-ptr->kcri-min.mini-val.dvi-int);
        *one-conj-flag = TRUE;
        break;

    'Greater than All Operator':
        perform One-Conjunction(kc-ptr->kcri-max.maxi-val.dvi-int);
        *one-conj-flag = TRUE;
        break;

  end case
end proc
```

```
proc N-Conjunction()

 /*  This procedure accomplishes the following:                  */
 /*  (1) Builds an N conjunction ABDL request using a template   */
 /*      (the unfinished return) provided by KMS.                */
 /*  (2) This is accomplished by loading in the action portion of */
 /*      the template, loading up to NUM-CONJ conjunctions into  */
 /*      the request, 'oring' the conjunctions together and then */
 /*      adding the target list.                                 */
 /*  (3) The conjunction portion is formed by copying from the   */
 /*      beginning of the conjunction to first asterik of the    */
 /*      template into ari request.  Then the next value from the */
 /*      the current file is inserted into the ari request,      */
 /*      followed by the remainder of the conjunction.  If this  */
 /*      is not the last conjunction of the request, then an or  */
 /*      is inserted and the next conjunction is constructed using */
 /*      the same process.                                       */

 begin proc

  abdl-ptr = sql-ptr->si-abdl-tran->ti-curr-req.ri-ab-req;
    /* Set pointer to the current ABDL request */

  if (kc-ptr->kcri-req-status == FIRSTTIME)
    /* Calculates values that will be used on all calls to this procedure */
    /* for a given request.  Thus, these values are only calculated once. */

    begin if
      for (i = 0; kc-ptr->kcri-unfin-ret[i] != LPARAN; i++)
        ;
      kc-ptr->kcri-beg-conj = i;/*Mark position where the conjunction begins*/
      action-len = kc-ptr->kcri-beg-conj;/*does not include 1st LPARAN */
      for ( ; kc-ptr->kcri-unfin-ret[i] != ASTERIK; i++)
        ;
      kc-ptr->kcri-beg-asterik = i;/*Mark position of the first asterik*/
```

```
    /* Calculates the size of the template. */
    unfin-ret-len = strlen(kc-ptr->kcri-unfin-ret);
    for (i = unfin-ret-len; kc-ptr->kcri-unfin-ret[i] != ASTERIK; i--)
      ;
    kc-ptr->kcri-end-asterik = i;/*Mark position of last asterik*/
    for ( ; kc-ptr->kcri-unfin-ret[i] != LPARAN; i++)
      ;
    for ( ; kc-ptr->kcri-unfin-ret[i] != RPARAN; i--)
      ;
    kc-ptr->kcri-end-conj = i;
      /*Mark position of the end of the conjunction*/
    target-len = unfin-ret-len - kc-ptr->kcri-end-conj + 1;
    conj-len = unfin-ret-len - target-len - action-len;

    /* Calculates the maximum length of the finished request. */
    kc-ptr->kcri-req-len = (action-len + (NUM-CONJ * conj-len) +
      (NUM-CONJ * ORLen) + target-len);
    kc-ptr->kcri-files.nri-curr.fi-fid =
      fopen(kc-ptr->kcri-files.nri-curr.fi-fname, "r");
    kc-ptr->kcri-req-status = RESTTIME;

  end if
else
  begin else
    /* Reset the length of the unfinished request */
    unfin-ret-len = strlen(kc-ptr->kcri-unfin-ret);
  end else

/* Allocates space for the finished request. */
abdl-ptr->ari-req = var-str-alloc(kc-ptr->kcri-req-len);

/* load request with action portion and an ( */
for (i = 0; i != (kc-ptr->kcri-beg-conj + 1); i++)
  abdl-ptr->ari-req[i] = kc-ptr->kcri-unfin-ret[i];
j = i;
```

```
counter = 1;
while ((counter <= NUM-CONJ) && (kc-ptr->kcri-num-values-cfile != 0))
/*Keep building conjunctions, filling them with values
  & 'oring' them together*/

  begin while
    /* loads template up to asterik */
    for (i = kc-ptr->kcri-beg-conj; i != kc-ptr->kcri-beg-asterik; i++)
      begin for
        abdl-ptr->ari-req[j] = kc-ptr->kcri-unfin-ret[i];
        j++;
      end for

    /* loads in the next value */
    for (i = 0; ((abdl-ptr->ari-req[j] =
            getc(kc-ptr->kcri-files.nri-curr.fi-fid)) != '0'); i++)
      j++;

    /* loads the appropriate number of ) behind the conj */
    for (i = (kc-ptr->kcri-end-asterik + 1);
          i != (kc-ptr->kcri-end-conj + 1); i++)
      begin for
        abdl-ptr->ari-req[j] = kc-ptr->kcri-unfin-ret[i];
        j++;
      end for

    if ((counter != NUM-CONJ) && (kc-ptr->kcri-num-values-cfile != 1))
      /* It is not the last conjunction */

      begin if
        /* loads " or " into the request to connect the conjs */
        abdl-ptr->ari-req[j++] = BLANKSPACE;
        abdl-ptr->ari-req[j++] = 'o';
        abdl-ptr->ari-req[j++] = 'r';
        abdl-ptr->ari-req[j++] = BLANKSPACE;
      end if
```

```
    else  /* It is the last conjunction */

      begin else
        /* loads the target list one value oer line */
        for (i = (kc-ptr->kcri-end-conj); i != (unfin-ret-len + 1); i++)
          begin for
            abdl-ptr->ari-req[j] = kc-ptr->kcri-unfin-ret[i];
            j++;
          end for

        /* checks if there is only one value in this request.        */
        /* if true then one set of parenthesis is replaced with blanks */
        if (counter == 1)
          begin if
            abdl-ptr->ari-req[kc-ptr->kcri-beg-conj] = BLANKSPACE;
            abdl-ptr->ari-req[kc-ptr->kcri-end-conj] = BLANKSPACE;
          end if

      end else

    counter++;
    kc-ptr->kcri-num-values-cfile--;

  end while

 if (kc-ptr->kcri-num-values-cfile == 0)
   /* Set the status to signify the last subrequest */
   sql-ptr->si-subreq-stat = LASTSUBREQ;

end proc
```

```
proc Not-In-Conjunction()

  /*  This procedure accomplishes the following:                    */
  /*  (1) Builds a one conjunction ABDL request using a template    */
  /*      provided by KMS.                                          */
  /*  (2) This is accomplished by loading in the action portion of  */
  /*      the template, loading up to NUM-CONJ conjunctions into    */
  /*      the request, 'anding' the conjunctions together and then  */
  /*      adding the target list.                                   */
  /*  (3) The conjunction portion is formed by copying from the     */
  /*      beginning of the conjunction to first asterik of the      */
  /*      template into ari request.  Then the next value from the  */
  /*      the current file is inserted into the ari request,        */
  /*      followed by the remainder of the conjunction.  If this    */
  /*      is not the last conjunction of the request, then an and   */
  /*      is inserted and the next conjunction is constructed       */
  /*      using the same process.                                   */

  begin proc

    abdl-ptr = sql-ptr->si-abdl-tran->ti-curr-req.ri-ab-req;
      /* Set pointer to the current ABDL request */

    if (kc-ptr->kcri-req-status == FIRSTTIME)
      /* Calculates values that will be used on all calls to this for a */
      /* given request.  Thus, these values are only calculated once. */

      begin if
        for (i = 0; kc-ptr->kcri-unfin-ret[i] != ASTERIK; i++)
          ;
        kc-ptr->kcri-beg-asterik = i;/*Mark position of first asterik*/
        for ( ; kc-ptr->kcri-unfin-ret[i] != LPARAN; i--)
          ;
        kc-ptr->kcri-beg-conj = i;/*Mark position where conjunction begins*/

        /* Calculates the size of the template */
        unfin-ret-len = strlen(kc-ptr->kcri-unfin-ret);
        for (i = unfin-ret-len; kc-ptr->kcri-unfin-ret[i] != ASTERIK; i--)
          ;
        kc-ptr->kcri-end-asterik = i;/*Mark position of the last asterik*/
        for ( ; kc-ptr->kcri-unfin-ret[i] != RPARAN; i++)
          ;
        kc-ptr->kcri-end-conj = i;
          /* Mark position where the conjunction ends */
```

```
    action-len = kc-ptr->kcri-beg-conj;
    target-len = unfin-ret-len - kc-ptr->kcri-end-conj;
    conj-len = unfin-ret-len - target-len - action-len;

    /* Calculates the maximum length of the finished request. */
    kc-ptr->kcri-req-len = (action-len + (NUM-CONJ * conj-len) +
      (NUM-CONJ * ANDLen) + target-len);
    kc-ptr->kcri-files.nri-curr.fi-fid =
      fopen(kc-ptr->kcri-files.nri-curr.fi-fname, "r");
    kc-ptr->kcri-req-status = RESTTIME;
  end if
else
  begin else
    /* Reset the length of the unfinished request */
    unfin-ret-len = strlen(kc-ptr->kcri-unfin-ret);
  end else

/* Allocates space for the finished request */
abdl-ptr->ari-req = var-str-alloc(kc-ptr->kcri-req-len);

/* load request with action portion and an (  */
for (i = 0; i != (kc-ptr->kcri-beg-conj); i++)
  abdl-ptr->ari-req[i] = kc-ptr->kcri-unfin-ret[i];
j = i;

counter = 1;
while ((counter <= NUM-CONJ) && (kc-ptr->kcri-num-values-cfile != 0))
  /* Keeps building conjunctions, filling them with values and . */
  /* 'anding' them together.                                     */

  begin while
    /* loads template up to asterik */
    for (i = kc-ptr->kcri-beg-conj; i != kc-ptr->kcri-beg-asterik; i++n)
      begin for
        abdl-ptr->ari-req[j] = kc-ptr->kcri-unfin-ret[i];
        j++;
      end for
```

```
    /* loads in next value */
    for (i = 0; ((abdl-ptr->ari-req[j] =
         getc(kc-ptr->kcri-files.nri-curr.fi-fid)) != '0'); i++)
      j++;

    /* loads a ) behind the conjunction */
    abdl-ptr->ari-req[j] = kc-ptr->kcri-unfin-ret[kc-ptr->kcri-end-conj];
    j++;

    if ((counter != NUM-CONJ) && (kc-ptr->kcri-num-values-cfile != 1))
    /* It is not the last conjunction */
      begin if
        /* loads " and " into the request to connect the conjs */
        abdl-ptr->ari-req[j++] = BLANKSPACE;
        abdl-ptr->ari-req[j++] = 'a';
        abdl-ptr->ari-req[j++] = 'n';
        abdl-ptr->ari-req[j++] = 'd';
        abdl-ptr->ari-req[j++] = BLANKSPACE;
      end if
    else /* It is the last conjunction */
      begin else
        /* loads the target list including a   */
        for (i = (kc-ptr->kcri-end-conj + 1); i != (unfin-ret-len + 1); i++)
          begin for
            abdl-ptr->ari-req[j] = kc-ptr->kcri-unfin-ret[i];
            j++;
          end for
      end else
    counter++;
    kc-ptr->kcri-num-values-cfile--;
  end while

 if (kc-ptr->kcri-num-values-cfile == 0)
   /* Set the status to signify the last subrequest */
   sql-ptr->si-subreq-stat = LASTSUBREQ;

end proc
```

```
proc One-Conjunction (value)

 /*  This procedure accomplishes the following:                    */
 /*  (1) Builds a one conjunction ABDL request using a template    */
 /*      provided by KMS.                                          */
 /*  (2) This is accomplished by loading in the unfinished return  */
 /*      up to the first asterik, loading in the value passed to   */
 /*      the procedure and then loading in the remainder of the    */
 /*      of the unfinished return.                                 */

 begin proc

  abdl-ptr = sql-ptr->si-abdl-tran->ti-curr-req.ri-ab-req;
   /* Set pointer to the current ABDL request */

  for (i = 0; kc-ptr->kcri-unfin-ret[i] != ASTERIK; i++)
   ;
  kc-ptr->kcri-beg-asterik = i; /* Mark postion of the first asterik */

  /* Calculate the maximum length of the finished request. */
  kc-ptr->kcri-req-len = (strlen(kc-ptr->kcri-unfin-ret) + INTSIZE);
  for (i = kc-ptr->kcri-req-len; kc-ptr->kcri-unfin-ret[i] != ASTERIK; i--)
   ;
  kc-ptr->kcri-end-asterik = i; /* Mark the position of the last asterik. */
  kc-ptr->kcri-files.nri-curr.fi-fid =
   fopen(kc-ptr->kcri-files.nri-curr.fi-fname, "r");

  /* Allocate space for the finished result. */
  abdl-ptr->ari-req = var-str-alloc(kc-ptr->kcri-req-len);

  /* load request up to the first asterik  */
  for (i = 0; i != kc-ptr->kcri-beg-asterik; i++)
   abdl-ptr->ari-req[i] = kc-ptr->kcri-unfin-ret[i];
  j = i;

   /* loads in the min or max value */
  num-to-str (value, value-str);
  for (k = 0; k != strlen(value-str); k++)
   begin for
    abdl-ptr->ari-req[j] = value-str[k];
    j++;
   end for
```

```
  /* loads the remainder of the request including the target list */
  for (i = (kc-ptr->kcri-end-asterik + 1);
        i != strlen(kc-ptr->kcri-unfin-ret) +1; i++)
    begin for
      abdl-ptr->ari-req[j] = kc-ptr->kcri-unfin-ret[i];
      j++;
    end for
  abdl-ptr->ari-req[j] = ' ':
  sql-ptr->si-subreq-stat = LASTSUBREQ;

end proc


proc chk-if-last-response()

 /* This procedure accomplishes the following:                    */
 /* (1) Determines the length of the response.                    */
 /* (2) Determines if this is the last response to a given request and   */
 /*     returns a boolean indicating such.                        */

 begin proc

  /* Calculates response length */
  for (response-length = 0;
     sql-ptr->si-kfs-data.kfsi-rel.kri-response[response-length] != EOResult;
     response-length++);
    ++response-length;

  /* Checks if this is the last response */
  if (sql-ptr->si-kfs-data.kfsi-rel.kri-response[response-length - 3]
      == CSignal)
    return(TRUE);
  else  /* It is not the last response */
    return(FALSE);

 end proc
```

```
proc file-future-results()

 /*  This procedure accomplishes the following:          */
 /*  (1) Removes the attribute names from the response.  */
 /*  (2) Places the remaining attribute values into the  */
 /*      future-results-file.                            */
 /*  (3) Keeps track of how many sub-requests there are. */
 /*  (4) Calculates and stores max and min values.       */

 begin proc

  max-ptr = &(kc-ptr->kcri-max);  /* Initialize pointer */
  min-ptr = &(kc-ptr->kcri-min);  /* Initialize pointer */
  f-ptr = &(kc-ptr->kcri-files);  /* Initialize pointer */

  if (kc-ptr->kcri-file-status == FIRSTTIME)
    /* Do the following initialization */

    begin if
      max-ptr->maxi-val.dvi-int = MINVAL;
      min-ptr->mini-val.dvi-int = MAXVAL;
      f-ptr->nri-futr.fi-fid = fopen(f-ptr->nri-futr.fi-fname, "w");
      kc-ptr->kcri-file-status = RESTTIME;
    end if

  else
    f-ptr->nri-futr.fi-fid = fopen(f-ptr->nri-futr.fi-fname, "a");

  kc-ptr->kcri-curr-pos = 1;
  response = sql-ptr->si-kfs-data.kfsi-rel.kri-response;
  kc-ptr->kcri-res-len = strlen(response);

  /* Number of values in the returned result of the request */
  num-values = &(kc-ptr->kcri-num-values-ffile);
```

```
while ( kc-ptr->kcri-curr-pos < (kc-ptr->kcri-res-len) - 2)

  begin while
    for (;response[kc-ptr->kcri-curr-pos] != EMARK;kc-ptr->kcri-curr-pos++)
      ; /* Skip the attribute name */
    (kc-ptr->kcri-curr-pos)++;
    for (val-len = 0;response[kc-ptr->kcri-curr-pos + val-len] != EMARK;
        val-len ++)
      ;  /* Find out how long the attribute value is */

    /* Allocate storage space for the value */
    temp-str = var-str-alloc(val-len + 1);
    j = 0;

  for ( ;response[kc-ptr->kcri-curr-pos] != EMARK;kc-ptr->kcri-curr-pos++)

      begin for
        /* Load the value into the future file */
        putc(response[kc-ptr->kcri-curr-pos], f-ptr->nri-futr.fi-fid);
        /* Load the value into the temp string */
        temp-str[j++] = response[kc-ptr->kcri-curr-pos];
      end for

    (kc-ptr->kcri-curr-pos)++;
    putc('0, f-ptr->nri-futr.fi-fid);
    temp-str[j] = ' ';
    *num-values = *num-values + 1; /* Count the number of values */

    /* Calculates the maximum value of those values returned so far */
    max-ptr->maxi-val.dvi-int =
            max(max-ptr->maxi-val.dvi-int, str-to-num(temp-str));

    /* Calculates the minimum value of those values returned so far */
    min-ptr->mini-val.dvi-int =
            min(min-ptr->mini-val.dvi-int, str-to-num(temp-str));
    free(temp-str);
  end while

 fclose(f-ptr->nri-futr.fi-fid);

end proc
```

```
proc swap-files()

 /* This procedure swaps the contents of the future file with the */
 /* contents of the current file.                                 */

 begin proc

  curr-fp = &(kc-ptr->kcri-files.nri-curr);
  fut-fp = &(kc-ptr->kcri-files.nri-futr);
  temp.fi-fid = curr-fp->fi-fid;
  strcpy(temp.fi-fname, curr-fp->fi-fname);
  curr-fp->fi-fid = fut-fp->fi-fid;
  strcpy(curr-fp->fi-fname, fut-fp->fi-fname);
  fut-fp->fi-fid = temp.fi-fid;
  strcpy(fut-fp->fi-fname, temp.fi-fname);

 end proc
```

```
module kfs ()

/*  This procedure accomplishes the following:                    */
/*  (1) Calls initialize()                                        */
/*  (2) Calls fill-table-headings()                               */
/*  (3) Calls one-hscreen-results() if the width of the           */
/*      output table is less than the width of the screen         */
/*  (4) Calls more() to output the results file if the last       */
/*      response buffer has been received                         */
/*  (5) Calls finish() to free used memory after the last         */
/*      response buffer has been received                         */

 begin module
  proc initialize(); /* Set up structures and variables for processing */
  proc fill-table-headings(); /* Get headings for relational output */
  if (table-width <= OutputCols) /* If table size <= screen width */
   proc one-hscreen-results();
  else
   proc all-hscreen-results(); /* This procedure has not been written */
  if (last response buffer)
   begin
    proc more(); /* Output relational output file  to screen */
    proc finish(); /* Close out sturctures and variables and free space */
   end if
 end module
```

```
proc initialize()

/*  This procedure accomplishes the following:                    */
/*  (1) Sets kfs-r-ptr to the address of the current relational   */
/*      database                                                  */
/*  (2) Sets kri-curr-pos to 1, the starting point in the response */
/*      buffer                                                    */
/*  (3) If this is the first time for a particular set of responses */
/*      then an Output File is opened for write status; otherwise  */
/*      the Output File is opened for append status               */


 begin proc
   set kfs-r-ptr to the address of the current relational database;
   kfs-r-ptr->kri-curr-pos = 1; /* Sets a pointer in the response array to
                                   the beginning of the array */
   if (kfs-r-ptr->kri-status == FIRSTIME) /* If this is the first time
                                             that this procedure has been
                                             called for this particular
                                             response.....then.....  */
     begin
       open Output File for "write" status;
       kfs-r-ptr->kri-status = FIRSTBUF; /* Change status to indicate that
                                            the FIRST BUFFER is being
                                            handled  */
     end if
   else
     open Output File for "append" status; /* This is not the first time
                                              thru this procedure so we
                                              need to append the results
                                              to the results already in the
                                              Output File */
 end proc
```

```
proc fill-table-headings()

/* This procedure accomplishes the following:                    */
/* (1) Fills different fields in various structures so that an   */
/*     output table similiar to one created in SQL can be made   */
/*     to show results coming from MBDS                          */

 begin proc
  curr-pos = kfs-r-ptr->kri-curr-pos; /* curr-pos used to hold actual
                                         current position in the results
                                         buffer so that kri-curr-pos can
                                         be set back to this value when we
                                         exit this procedure */
  allocate a new table-entry-info node;
  read the attribute name from the response buffer;
  determine the length of this name and store in new-tei-ptr->tei-name-len;
  new-tei-ptr->tei-val-len = proc get-size(new-tei-ptr->tei-attr);
    /* Get the max size that this attr can possibly take on */
  new-tei-ptr->tei-col-len = proc max(new-tei-ptr->tei-name-len,
                                      new-tei-ptr->tei-val-len);
     /* Determine the actual column size in the output table */
  thi-first-ent = new-tei-ptr; /* First entry for output table is equal
                                  to the results we just obtained */
  thi-curr-ent = new-tei-ptr; /* The current entry is also equal to these
                                 results */
  proc skipnameorval(); /* Skip over the attr value in response buffer
                           until the next attr name is hit */
  temp-attr = next attr name in response buffer;
  while (temp-attr <> thi-first-ent->attr-name)
    begin
      allocate a new table-entry-info node; /*i.e., new-tei-ptr */
      new-tei-ptr->tei-attr = temp-attr;
      determine the length of this name and store
        in new-tei-ptr->tei-name-len;
      new-tei-ptr->tei-val-len = proc get-size(new-tei-ptr->tei-attr);
          /* Get the max size that this attr can possibly take on */
      new-tei-ptr->tei-col-len = proc max(new-tei-ptr->tei-name-len,
                                          new-tei-ptr->tei-val-len);
         /* Determine the actual column size in the output table */
      tbl-ptr->thi-curr-ent->tei-next = new-tei-ptr;
      tbl-ptr->thi-curr-ent = new-tei-ptr;
      proc skipnameorval();
      temp-attr = next attr name in response buffer;
    end while
  kfs-r-ptr->kri-curr-pos = curr-pos; /* Restore current position */
end proc
```

```
proc one-hscreen-results()

/* This procedure outputs the results in SQL table form if this table */
/* can fit within the width of one screen                             */

 begin proc
  if (FIRSTBUF == TRUE)
    begin
      proc load-titles(); /* Output the headings of the table */
      kfs-r-ptr->kri-status = RESTBUFS; /* Change status to indicate
                                   that titles/headings no
                                   longer have to be output */
    end if
  while (kfs-r-ptr->kri-curr-pos < kfs-r-ptr->kri-res-len)
    begin
      tbl-ptr = kfs-r-ptr->kri-form-data.thi-first-ent;
        /* Get the first table entry so that you can work from here */
      while (tbl-ptr <> NULL)
        begin
          proc skipnameorval();
          column-difference = tbl-ptr->tei-col-len - proc get-val-len();
            /* column-difference indicates the difference between the
               actual column length and the length of the attr value to
               be output.  We need this to determine how many spaces are
               needed to keep our results left-justified */
          print the attr value;
          print a series of blank spaces equal to the column-difference;
          print a "|";
          tbl-ptr = tbl-ptr->tei-next; /* Get the next table entry */
        end while
      print a carriage return;
    end while
    close Output File;
 end proc
```

```
proc load-titles()

/* This procedure loads the heading of a SQL results table into the */
/* output file                                                      */

 begin proc
  tbl-ptr = kfs-r-ptr->kri-form-data.thi-first-ent;
    /* Get the first table entry so that you can work from here */
  while (tbl-ptr <> NULL)
   begin
    column-difference = tbl-ptr->tei-col-len - tbl-ptr->tei-name-len;
         /* column-difference indicates the difference between the
           actual column length and the length of the attr value to
           be output.  We need this to determine how many spaces are
           needed to keep our results left-justified */
    print the attr name;
    print a series of blank spaces equal to the column-difference;
    print a "|";
    tbl-ptr = tbl-ptr->tei-next;
   end while
  print a carriage return;
  print a series of "-" equal to the width of the table;
 end proc


proc get-size(x)

/* This procedure obtains the maximum size that a particular attribute */
/* value may take on                                                   */

 char x;

 begin proc

  traverse the table entry list until you find the attr name equal to x;
  return(the length of this attr name);
 end proc
```

```
proc more()

/*  This procedure is just like the more facility offered in Unix */
/*  It is not as sophisticated, however.                          */

 begin proc
  open kfs-r-ptr->kri-o-file.fi-fname for "read" status;
  get a char from opened file;
  while (NOT EOF)
    begin
     counter = 0; /* counter is used to keep track of how many lines
                     have been printed on the screen */
     while ((counter <= screen-heigth) AND (NOT EOF))
       begin
         print the char;
         if (c == carriage return)
           counter = counter + 1;
         get a char from opened file;
       end while
     if (counter > screen-heigth)
       begin
         print the word "--more--";
         determine if user wants to quit or advance 1 to screen-heigth lines
                in the opened file;
       end if
    end while
 end proc

proc skipnameorval()

/*  This procedure skips over either an attribute name or attribute value */
/*  depending where kri-curr-pos is currently located.  This is necessary */
/*  because results are coming back as:  ATTR-NAME  ATTR-VALUE.  In some  */
/*  situations we want just the NAME and in others we want just the VALUE */

 begin proc

   update kri-curr-pos to skip over the the attr name or value
       that it is currently positioned at;

 end proc
```

```
proc finish()

/*  This procedure frees any structure space that may have been created */
/*  during the creation of the ouput table                              */

 begin proc
  tbl-ptr = current relational database;
  tbl-ent-ptr = tbl-ptr->thi-first-ent;/* Set tbl-ent-ptr to the first
                                        table entry /*
  while (tbl-ent-ptr <> NULL)
   begin
    tbl-ptr->thi-first-ent = tbl-ent-ptr->tei-next;
      /* Get the next table entry */
    tbl-ent-ptr->tei-next = NULL;
    free(tabl-ent-ptr);
    tbl-ent-ptr = tbl-ptr->thi-first-ent;
   end while
 end proc
```

APPENDIX F - THE SQL USERS' MANUAL

A. Overview

The SQL language interface allows the user to input transactions from either a file or the terminal. A transaction can take the form of either creates of a new database or queries against an existing database. The SQL language interface is menu-driven. When the transactions are read from either a file or the terminal they are stored in the interface. If the transactions are creates they are executed automatically by the system. If the transactions are queries the user will be prompted by another menu to selectively pick an individual query to be processed. The menus provide an easy and efficient way to allow the user to see and select the methods in which to perform the mapping functions. Each menu is tied to its predecessor so that by exiting each menu the user is moved back up the menu "tree". This allows the user to perform multiple tasks in one session.

B. USING THE SYSTEM

There are two operations the user can perform on the database schemas. The user can either create a new database or process queries against an existing database. The first menu displayed prompts the user for which function to perform. This menu, hereafter named MENU1, looks like the following:

```
Enter type of operation desired
     (l) - load a new database
     (p) - process old database
     (x) - return to the operating system

ACTION ----> _
```

Upon selecting the desired operation, the user will be prompted to enter the name of the database to be used. For the case that the load operation was selected, the database name provided cannot be presently used. Likewise, for a process old operation the database name provided must be in existence. In either case, if an error occurs the user will be told to rekey a different name. The session continues once a valid name has been entered.

For either type of operation selected from MENU1, the second menu is the same and asks for the mode of input. This input may come from a data file or interactively from the terminal. The generic menu, called MENU2, looks like the following:

```
Enter mode of input desired
     (f) - read in a group of transactions from a file
     (t) - read in transactions from the terminal
     (x) - return to the previous menu

ACTION ----> _
```

If the user wishes to read transactions from a file he will be prompted to provide the name of the file that contains those transactions. If the user wishes to enter

transactions directly from the terminal a message will be displayed reminding him of the correct format and special characters that must be used. Since the transaction list stores both creates and queries, two different access methods must be employed to send the two types of transactions to the KMS. Therefore, our discussion branches to handle the two processes the user will encounter.

1. Processing Creates

When the user has specified the filename of creates (if the input is from a file) or typed in a set of creates (if the input is from the terminal), further user intervention is not required. It does not make sense to process only a single create out of a set of creates that produce a new database since they all must be processed at once and in a specific order. Therefore, the transaction list of creates is automatically executed by the system. Since all the creates must be sent at once to form a new database, control should not return to MENU2 where further transactions can be input. Instead, control returns to MENU1 where the user can pick a new operation or new database.

2. Processing Queries

In this case, after the user has specified his mode of input, he will conduct an interactive session with the system. First, all queries will be listed to the

screen. As the queries are listed from the transaction list, a number is assigned to each query in ascending order starting with the number one. The number is printed on the screen beside the first line of each query. Next, an access menu, called MENU3, is displayed which looks like the following:

```
Pick the number or letter of the action desired
     (num) - execute one of the preceding queries
     (d)   - redisplay the list of queries
     (x)   - return to the previous menu

ACTION ----> _
```

Since the displayed queries might exceed the vertical height of the screen, only a screen full of queries will be displayed at one time. If the desired query is not on the current page, the user can hit the RETURN key to display the next page of queries. If the user only wishes to print a certain number of lines, then after the first page is displayed the user can enter a number and only that many lines of queries will be displayed. If the user is only looking for certain queries, once he has found them he does not have to page through the entire transaction list. By hitting the q key, control will break from listing queries and MENU3 will be displayed. Under normal conditions when the end of the transaction list has been viewed, MENU3 will appear.

Since queries are independent items, the order in which they are processed does not matter. The user has the choice of executing however many queries he desires. A loop causes the query listing and MENU3 to be redisplayed after any query has been executed so that further choices may made. Unlike processing creates, control returns to MENU2 because the user may have more than one file of queries against a particular database or he may wish to input some extra queries directly from the terminal. Once he has finished processing on this particular database, he can exit back to MENU1 to either change operations or exit to the operating system.

C. DATA FORMAT

When reading transactions from a file or the terminal, there must be some way of distinguishing when one transaction ends and the next begins. Transactions are allowed to span multiple lines as evidenced by a typical nested SQL select. Since the system is reading the input line by line, an end-of-transaction flag is needed. In our system this flag is the "@" character. Likewise, the system needs to know when the end of the input stream has been reached. In our system the end-of-file flag is represented by the "$" character. The following is an example of an input stream with the necessary flags that must be included when multiple transactions are entered:

```
TRANSACTION #1
@
TRANSACTION #2
@
.
.
.

@
TRANSACTION #n
$
```

D. RESULTS

When the results of the executed transactions are sent back to the user's screen, they will be displayed exactly the same way queries are displayed (See section B-2). The following consolidates the user's options:

KEY	FUNCTION
return	displays next screenful of output
(number)	displays only (number) lines of output
q	stops output, MENU1 is then redisplayed

LIST OF REFERENCES

(1) Demurjian, S. A. and Hsiao, D. K., "New Directions in Database-Systems Research and Development," in the Proceedings of the New Directions in Computing Conference, Trondheim, Norway, August, 1985; also in Technical Report, NPS-52-85-001, Naval Postgraduate School, Monterey, California, February 1985.

(2) Macy, G., Design and Analysis of an SQL Interface for a Multi-Backend Database System, M. S. Thesis, Naval Postgraduate School, Monterey, California, March 1984.

(3) Rollins, R., Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.

(4) Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files," Communications of the ACM, Vol. 13, No. 2, February 1970, also in Corrigenda, Vol 13., No. 4, April 1970.

(5) Wong, E., and Chiang, T. C., "Canonical Structure in Attribute Based File Organization," Communications of the ACM, September 1971.

(6) Rothnie, J. B. Jr., "Attribute Based File Organization in a Paged Memory Environment," Communications of the ACM, Vol. 17, No. 2, February 1974.

(7) The Ohio State University, Columbus, Ohio, Technical Report No. OSU-CISRC-TR-77-7, DBC Software Requirements for Supporting Relational Databases, by J. Banerjee and D. K. Hsiao, November 1977.

(8) Naval Postgraduate School, Monterey, California, Technical Report, NPS52-85-002, A Multi-Backend Database System for Performance Gains, Capacity Growth and Hardware Gains, by S. A. Demurjian, D. K. Hsiao and J. Menon, February 1985.

(9) Astrahan, M. M., et al., "System R: Relational Approach to Database Management," ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976.

(10) Boehm, B. W., Software Engineering Economics, Prentice-Hall, 1981.

(11) Naval Postgraduate School, Monterey, California, Technical Report, NPS52-84-012, Software Engineering

Techniques for Large-Scale Database Systems as Applied to the Implementation of a Multi-Backend Database System, by Ali Orooji, Douglas Kerr and Daivid K. Hsiao, August 1984.

(12) The Ohio State University, Columbus, Ohio, Technical Report No. OSU-CISRC-TR-82-1, The Implementation of a Multi Backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS, by D. S. Kerr et al, January 1982.

(13) Kernighan, B. W., and Ritchie, D. M., The C Programming Language, Prentice-Hall, 1978.

(14) Howden, W. E., "Reliability of the Path Analysis and Testing Strategy," IEEE Transactions on Software Engineering, Vol. SE-2, September 1976.

(15) Johnson, S. C., Yacc: Yet Another Compiler-Compiler, Bell Laboratories, Murray Hill, New Jersey, July 1978.

(16) Lesk, M. E. and Schmidt, E., Lex - A Lexical Analyzer Generator, Bell Laboratories, Murray Hill, New Jersey, July 1978.

(17) Date, C. J., An Introduction to Database Systems, 3d ed., Addison Wesley, 1982.

(18) Shienbrood, E., More - A File Persual Filter for CRT Viewing, Bell Laboratories, Murray Hill, New Jersey, July 1978.

(19) Benson, T. P. and Wentz, G. L., The Design and Implementation of a Hierarchial Interface for the Multi-Lingual Database System M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
4.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5100	1
5.	Professor David K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	1
6.	Steven A. Demurjian, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
7.	Gary R. Kloepping Route 1, Box 99 Santa Rosa, Texas 78593	3
8.	John F. Mack 2934 Emory Street Columbus, Georgia 31903	3
9.	Tim P. Benson P. O. Box 1974 Woodbridge, Virginia 22193	2
10.	Gary L. Wentz 111 Appian Way Pasadena, Maryland 21122	2